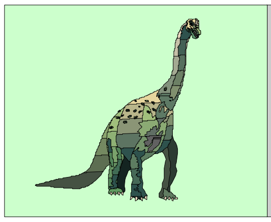


The Software Survivors

Relating Software Survival to the Evolution of Complexity

by W. Cave, & R. Wassmer
Visual Software International, May 15, 2014

BACKGROUND

To learn from history, one must be a seeker of truth - even when the results of scientific experiments go against the biases of one's beliefs. When comparing differences in thinking across widespread segments of the global population, one may conclude that *Truth is Key to Survival*.

The February '96 issue of Upside magazine compared changes in technology to the evolution of mankind resulting from the "Cambrian Explosion" on planet Earth, [1]. Before this event, no molecules were sufficiently complex for evolutionary processes to proceed to the species we have today. The article noted that explosions in technology follow similar trends.

The industrial revolution developed machinery to levels of complexity well beyond anything less than a century before. Scientific revolutions in atomic physics, aerospace, and electronics broke down major barriers of complexity. Computer-Aided Design (CAD) led to explosions in semiconductor chip technology, with engineers achieving complex designs of the integrated circuit and Personal Computer (PC). PCs now allow individuals to solve complex problems previously limited to large mainframes.

So what is the underlying motivation? According to W. Edwards Deming, a leader in the field of quality control, *happiness is constant improvement*, [2]. To be happy, one must see constant improvement in the quality of life. When most people are working to improve their lives, those "standing still" appear to be falling behind. In today's technological world, improvement in our lives is often accomplished through improvements in technology.

Looking at the history of technology, breakthroughs make life easier by solving problems of significant complexity. One can conclude that, *to achieve happiness, we must be able to deal with growing levels of complexity*. So where is the next technological breakthrough likely to occur? In our view, it should be in software.

A HISTORY OF THE PROBLEM

During the 1960s and 1970s, software productivity was increasing by all measures, with articles published on ways to improve programming languages. The principle languages of the time were FORTRAN and COBOL. Various studies showed that COBOL had grown to support 80% of the world's software applications. Combined with FORTRAN, they occupied about 95%. But getting there was a fight. Assembly language programmers argued that both of these languages were too slow, even though scientific experiments proved them wrong. In those days managers came up through the programming ranks, so truth and economics won out. High school programmers replaced PhDs, and time and cost of development and support dropped dramatically. So how did we get to where we are?

During the 1980s, software studies showed productivity turning downhill, [3], [4], [5]. This followed the decision by AT&T to compete with IBM in computers. Billions were spent to promote the UNIX operating system, and a language called C went along for the ride. Wall Street - invested in UNIX companies like SUN - pressured their data processing departments to move to the new boxes. IBM lost major clients. When transaction processing times shot way up, customers returned to main frames in what became known as the "UNIX After Market."

About this same time, along came the PC. Small Silicon Valley companies started to sell computers and software to small businesses that heretofore could not afford them. This miracle soon spread to individuals and academics as well as larger businesses.

Although FORTRAN made it easy for engineers to build complex mathematical systems, the commercial data processing world was the biggest user of computers. FORTRAN was not suitable for handling big data and did not require college level courses to learn. COBOL was shunned by engineers and academics because it did not support mathematical programming. Being easy to learn and use, it was akin to vocational training - not suitable for college courses. When C was upgraded with pointers to C^{++}, and Object Oriented Programming (OOP) came about, AT&T overwhelmed the literature with articles promoting its software. Universities used this esoteric approach to achieve dramatic expansion of their Computer Science schools.

Covering up the slowness of C, computer speeds followed an exponential rise, until about 2006 when clock rates hit a wall. Prior to that clock speeds were doubling every 18 months, doubling software speeds without any effort. But software systems that were "improved" led to (Nicklaus) Wirth's Law: *Software gets slower faster than hardware gets faster*. [6].

When clock rates leveled off at about 4 GHz, chip manufacturers started putting multiple processors (cores) on a chip, implying that 8 processors should yield a speed-up of 8. Nothing appears further from the truth. The trail of failed parallel processor manufacturers goes back to the 1970s. Although software developers are quick to blame hardware manufacturers, the real problem is the approach to building software.

In spite of what most people think, many leading technologists support the view that software productivity has been going down and failures have been going up. John Warnock, CEO of Adobe and Gordon Bell, architect of the DEC VAX family are quoted as predicting Object-Oriented Programming (OOP) to be a disappointing technology, [7]. Henry Ledgard, famous author on programming languages has written that it does not deliver real improvements in software productivity, [8]. Many engineers now agree that C-based languages (C, C^{++}, C$^{\#}$, Java, etc.) have added a significant burden of needless complexity to their software projects.

As described by industry spokesman Paul Strassmann, most software environments treat programming like a craft, lacking the necessary separation of skills common to an industrial environment, [9]. Trying to move to an industry has been difficult. Strassman makes a strong case that separation of skills is key to dealing with complexity and becoming a productive industry. This was well before the current press to use parallel processors.

Comments below describe the level of urgency toward solving the parallel processor software problem while talking about the number of years it may take based upon history:

Intel Corp.'s Chief Technology Officer (CTO), Justin Rattner: -- "As hardware technology approaches the terascale level on the desktop, software has fallen further behind." -- "One result has been a lack of parallel programming applications to leverage dual-and multi-core processing technology. Intel is looking for 'new languages for programming in parallel,' [11].

"The industry is in a little bit of a panic about how to program multi-core processors, ... " said Chuck Moore, CTO at Advanced Micro Devices (AMD) trying to rally support for work in the area. "To make effective use of multi-core hardware today you need a PhD in computer science." [12].

Craig Mundie, CTO at Microsoft, said this is the "tip of the iceberg." --- To maximize computing horsepower, software makers will need to change how software programmers work. Only a handful of programmers in the world know how to write software code to divide computing tasks into chunks that can be processed at the same time instead of a traditional, linear, one-job-at-a-time approach. --- A new programming language would be required, and could affect how almost every piece of software is written. --- "This problem will be hard," [13].

RESISTANCE TO NEW SOLUTIONS

Although apparent from the above statements, barriers to a new software technology must be broken. But separate books by Christensen, [14], and Kuhn, [15], show that disruptive technologies must overcome tremendous inertia caused by three factors:

- Job Security - People have invested years becoming experts in an existing technology. The new technology eliminates the need for their expertise.
- Not Invented Here - The NIH factor exists in government or academic research laboratories where people are paid large sums of money to come up with solutions.
- Financial Competition - Huge financial investments exist in the current technology and are at risk of being wiped out.

The above bullets are reminiscent of companies trying to sell robots to the U.S. Auto industry in the 1960s. Based upon union threats, they were not allowed on the shop floors in Detroit. So the robots were sold to Japan where manufacturers proceeded to automate their factories. Today, Detroit is in bankruptcy while the Japanese auto industry is a dominant force in the world, and building new plants in the U.S. With chip designers now being led by the programmers, could Silicon Valley be headed in the same direction?

Parnas, [16], Sitner, [17], and Yourdon, [18], among others have pointed out the lack of a scientific approach to measuring software productivity. Experimental results and evaluations of software technology fall far short of their counterpart benchmarks of hardware. Lack of experimental evidence and scientific deduction has led to misperceptions of programmer job security. Substantial evidence points to *perceived job security* as based on hidden knowledge of what lies behind complex code. But *real job security* in the software business must ultimately depend upon one's productivity in a competitive economic environment - worldwide. Software productivity must be analyzed in terms of understandability, independence and reusability of modules - the same measures used in a hardware environment.

The solution described below was developed in the U.S as a CAD simulation tool to support design of military communication systems. As it became known, manufacturers in NATO countries wanted it to build their own simulations. Programmers in those countries refused to work with it. They enjoyed the job protection of highly symbolic C-based languages. The end result was that they could match neither the fidelity nor the speed of the simulations. When management purchased the simulations as well as the software environment, programmers tried to rewrite the code in C^{++} and finally gave up.

When working with most software today, there is a clear lack of organization compared to typical scientific and engineering fields. Forcing an organization of people who are not amenable to it may be impossible. However, young people with little vested interest in current approaches enjoy being very productive - very fast. If that implies being organized, they are willing to do it, provided the system they use helps them to accomplish their goals faster.

REQUIREMENTS FOR A SOLUTION

Is there an equivalent "Cambrian Explosion" on the software horizon. Pulses of promise appeared over the past few decades. But these solutions are not based upon sound experimental science. There is reasonable agreement among people experienced in many languages that popular approaches to building software have *increased* the level of complexity faced by developers instead of decreasing it, particularly in the support and upgrade mode.

When reading history written by the developers from Bell Labs, [19], one finds that C was designed to quickly port a game played by a few researchers waiting to be assigned to a project. The design goals were made clear by the original designer, Ken Thompson: (1) Use a Spartan syntax to keep the compiler simple to write; and (2) Keep the compiler small to fit in the PDP 7's tiny memory. It was never intended to be a real programming language. The original version of C was upgraded a number of times, but still falls behind FORTRAN and COBOL in speed. Because of the poor handling of data in C, pointers were added to C^{++} to help speed things up. But this solved only a small part of the problem.

Referring to Peter van der Linden's book**,** Deep C Secrets on the use of C and C^{++} in a production environment at SUN, he questions the placement of the burden of translating code, [20]. *Should it be on the programmer, or on the language translator?* Placement of this burden is important for single processors, but becomes critical when using parallel processors.

Looking at the software horizon, inspiring new solutions exist. These evolved in a competitive engineering environment - one requiring measurable productivity improvements, [21]. Breaking down barriers of complexity is the critical direction required. This is best measured by understandability, leading to increased productivity. So what are the analogies that may be used to solve this problem?

Starting with the requirement to split software into modules that run concurrently on parallel processors, one must understand the property of module independence that allows this to occur. Simply stated, independent modules may not share data directly, [22]. This is an organizational problem. When looking at such problems, none is more representative than the military. Their solution is obvious: everything must be organized into hierarchies, from top to bottom. This does not exist in today's popular programming languages.

Given a language that simplifies representation of hierarchies, one only has to look at the layers of hierarchy required to describe the data spaces needed to simplify otherwise complex algorithms. This problem is analogous to picking the best space in which to solve a complex mathematical problem. As the spatial definition becomes more complex, the algorithms are simplified. But they must be organized into hierarchies that are easily understood. Current popular languages have none of the sophisticated facilities required to easily create and understand a multi-layered hierarchical approach.

Given that one can design deep multi-layered data hierarchies, the next requirement is to provide equivalent facilities for instruction hierarchies. Given these facilities in a language, it becomes much easier to develop independent modules and their special interfaces. When implemented properly, the result is dramatic improvements in speed - even on a single processor.

This requirement is paramount when working with parallel processors. Grouping hierarchies of data into separate subspaces that only support a particular module, or into separate subspaces that act as the minimal interface between two modules yields the required property of independence. Having organized the data spaces as described above leads to dramatic improvements in understanding how multiple modules work independently to support each other.

Many of these requirements were prevalent in the 1970s, but have since been forgotten. For example, one in-one out control structures were known to greatly simplify understanding. This required a language that supported the instruction hierarchies needed to accomplish this. Current popular languages remain within the waterfall category, i.e., no hierarchies.

A TESTED SOLUTION

A CAD approach has evolved based upon a language environment designed to support very complex applications on parallel processors, [22]. Using hierarchical structures to support both data and instructions, one can design software architectures with engineering drawings. Large hierarchical structures support *separation of data from instructions*. The CAD approach implementing this has separate languages for describing data structures and rule structures. The result is profound simplification, yielding large increases in understanding and productivity.

A second paradigm shift follows directly from the first. This is *the separation of architecture from language*, see Figure 1. After reviewing design rules for building independent modules with this CAD approach it becomes apparent that architecture is as important in software as in other engineering fields. It supports the separation of skills noted by Strassmann.

These new paradigms put software on a plane with hardware. Architecture provides visibility of design, from top to bottom. Using simple design rules, one can ensure independence of modules, cutting through complexity and leading to real reusability. Graphical depiction of architecture yields a one-to-one mapping to the language elements. Two distinct icon types are used to represent these elements - one for data structures and one for rules.

The architecture of independent modules can be related to distribution of a system's database into independent data structures relative to transformations (rules). This permits architects to group primitive elements - data icons and rule icons - into modules using design rules that maximize module independence.

Separate languages for data structures and rules support representation of hierarchical information structures and rule structures that are easily understood by system designers. One need not have programming experience to understand the details of a system implementation. Both size and scope of these primitive elements are visible from the engineering drawings.

Achieving a good balance between what should be depicted graphically versus what should be described in a language is clearly affected by the size and scope of the data and rule structures. The combination of graphics and natural language represents a dramatic improvement in productivity for designers who can then focus on understanding the complexities of an application, and not get lost in "quirky" details of a programming language.

Having engineering drawings to control complex software product life cycles supports geographically distributed development efforts. This includes the collaborative efforts of highly skilled individuals or teams distributed internationally as well as nationally.

The biggest improvements for management are visibility of the design from top to bottom, and clarity of the implementation details. These come at no cost to module isolation and protection, provided one follows the design rules to ensure module independence. To enforce the rules one simply inspects the engineering drawings.

Architectural restructuring is another area that benefits from this new approach. Since software is selected over hardware for flexibility, one must be able to change it easily to accommodate new features and functions. But most software is poured into concrete shortly after the first pieces of code are written. Then everyone watches in frustration as it evolves - out of control - into a typical "rat's nest."

Using the hierarchical paradigms, one can restructure both architecture and code easily - after as well as during - the development process. This helps to create modules that can support added functionality while maintaining independence. "Rats nesting" is suppressed, removing the exponential approach to project completion (always close but never there). This technique can cut development time by whole numbers, as one learns about new implementation details and the corresponding changes required for their support.

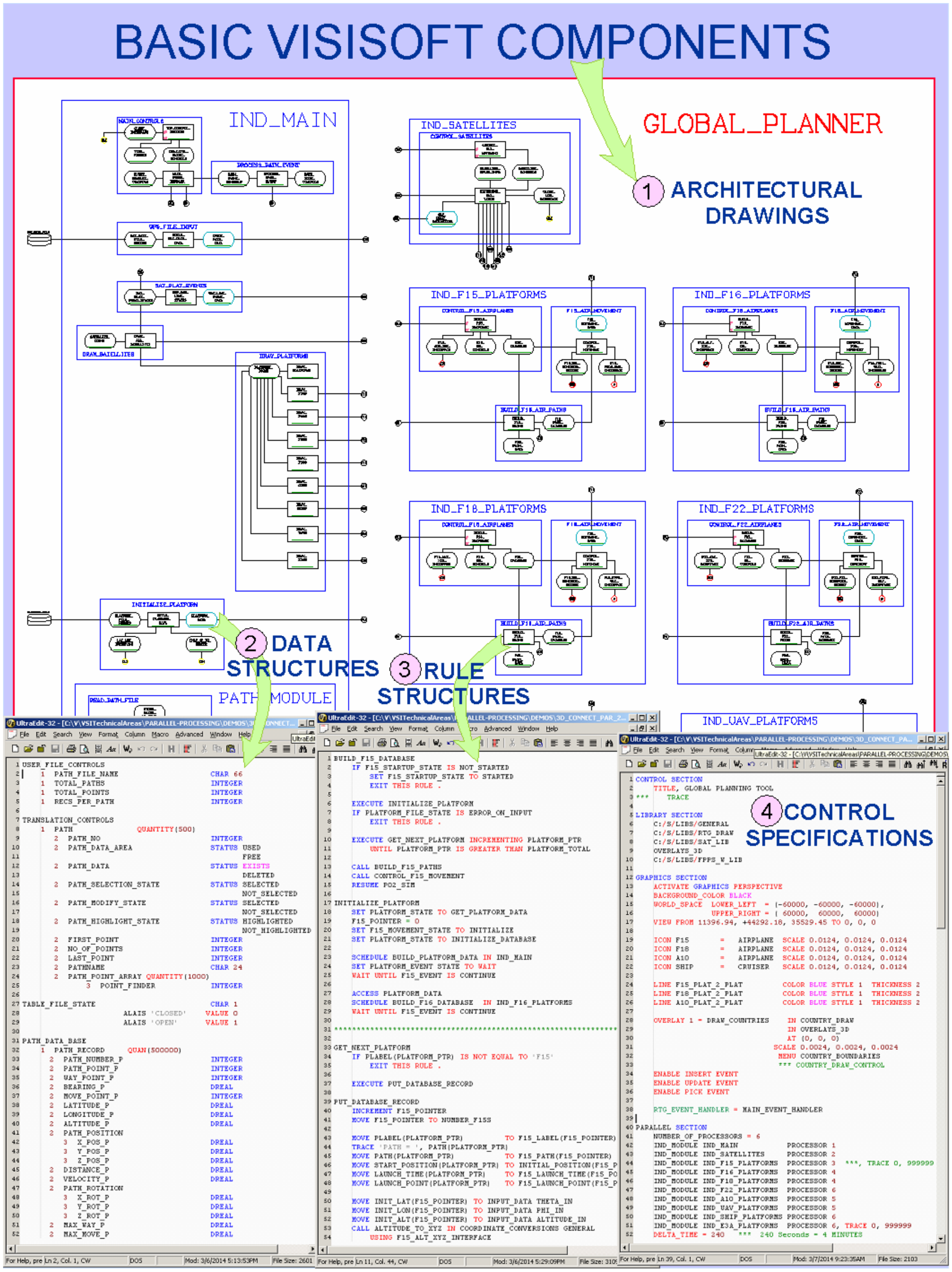
The CAD tool that implements these new paradigms is ***VisiSoft.*** Using this new approach, one can expect to deal with much higher levels of software complexity. At the same time, it can significantly reduce the number of people required to complete a project as well as reduce their level of programming expertise. Productivity can benefit most in the maintenance mode, particularly when major design changes are required to complex systems.

Underlying this system are three language translators as shown in Figure 1: one for data; one for instructions; and one for run-time control. Control Specifications render the software independent of the OS and platform, eliminating scripts while supporting complex databases and graphics. To make it easy for the human to understand, the languages are context oriented, requiring the translators to be extremely complex pieces of software.

SUMMARY

In a competitive economic environment, truth eventually surfaces in terms of time and cost savings. This is especially true when the differences in measures are significant. Improving productivity while maintaining a high level of quality for a given product will not be sufficient to survive in a free market society. This is like building a factory for one model of automobile. To survive in a competitive environment, one must be able to deal with changing models, ones that are increasing in complexity every year. As software products and users become more mature, this is the direction in which they must head.

This has been the direction taken by the developers of VisiSoft. The entire CAD system, including the languages, has evolved over many years of scrutiny and change. These changes will continue to ensure that the economic needs of the software market are met.



```
USER_FILE_CONTROLS
      1   PATH_FILE_NAME                          CHAR 66
      1   TOTAL_PATHS                             INTEGER
      1   TOTAL_POINTS                            INTEGER
      1   RECS_PER_PATH                           INTEGER

TRANSLATION_CONTROLS
      1   PATH                 QUANTITY(500)
          2   PATH_NO                             INTEGER
          2   PATH_DATA_AREA                      STATUS USED
                                                         FREE
          2   PATH_DATA                           STATUS EXISTS
                                                         DELETED
          2   PATH_SELECTION_STATE                STATUS SELECTED
                                                         NOT_SELECTED
          2   PATH_MODIFY_STATE                   STATUS SELECTED
                                                         NOT_SELECTED
          2   PATH_HIGHLIGHT_STATE                STATUS HIGHLIGHTED
                                                         NOT_HIGHLIGHTED
          2   FIRST_POINT                         INTEGER
          2   NO_OF_POINTS                        INTEGER
          2   LAST_POINT                          INTEGER
          2   PATHNAME                            CHAR 24
          2   PATH_POINT_ARRAY QUANTITY(1000)
                  3   POINT_FINDER                INTEGER

TABLE_FILE_STATE                                  CHAR 1
                       ALAIS 'CLOSED'     VALUE 0
                       ALAIS 'OPEN'       VALUE 1

PATH_DATA_BASE
      1   PATH_RECORD      QUAN(500000)
         2   PATH_NUMBER_P                        INTEGER
         2   PATH_POINT_P                         INTEGER
         2   WAY_POINT_P                          INTEGER
         2   BEARING_P                            DREAL
         2   MOVE_POINT_P                         INTEGER
         2   LATITUDE_P                           DREAL
         2   LONGITUDE_P                          DREAL
         2   ALTITUDE_P                           DREAL
         2   PATH_POSITION
             3   X_POS_P                          DREAL
             3   Y_POS_P                          DREAL
             3   Z_POS_P                          DREAL
         2   DISTANCE_P                           DREAL
         2   VELOCITY_P                           DREAL
         2   PATH_ROTATION
             3   X_ROT_P                          DREAL
             3   Y_ROT_P                          DREAL
             3   Z_ROT_P                          DREAL
         2   MAX_WAY_P                            DREAL
         2   MAX_MOVE_P                           DREAL
```

For Help, pre | Ln 2, Col. 1, CW | DOS | Mod: 3/6/2014 5:13:53PM | File Size: 2601

```
BUILD_F15_DATABASE
    IF F15_STARTUP_STATE IS NOT_STARTED
        SET F15_STARTUP_STATE TO STARTED
        EXIT THIS RULE .

    EXECUTE INITIALIZE_PLATFORM
    IF PLATFORM_FILE_STATE IS ERROR_ON_INPUT
        EXIT THIS RULE .

    EXECUTE GET_NEXT_PLATFORM INCREMENTING PLATFORM_PTR
        UNTIL PLATFORM_PTR IS GREATER THAN PLATFORM_TOTAL

    CALL BUILD_F15_PATHS
    CALL CONTROL_F15_MOVEMENT
    RESUME PO2_SIM

INITIALIZE_PLATFORM
    SET PLATFORM_STATE TO GET_PLATFORM_DATA
    F15_POINTER = 0
    SET F15_MOVEMENT_STATE TO INITIALIZE
    SET PLATFORM_STATE TO INITIALIZE_DATABASE

    SCHEDULE BUILD_PLATFORM_DATA IN IND_MAIN
    SET PLATFORM_EVENT STATE TO WAIT
    WAIT UNTIL F15_EVENT IS CONTINUE

    ACCESS PLATFORM_DATA
    SCHEDULE BUILD_F16_DATABASE  IN IND_F16_PLATFORMS
    WAIT UNTIL F15_EVENT IS CONTINUE

*************************************************************************

GET_NEXT_PLATFORM
    IF PLABEL(PLATFORM_PTR) IS NOT EQUAL TO 'F15'
        EXIT THIS RULE .

    EXECUTE PUT_DATABASE_RECORD

PUT_DATABASE_RECORD
    INCREMENT F15_POINTER
    MOVE F15_POINTER TO NUMBER_F15S

    MOVE PLABEL(PLATFORM_PTR)           TO F15_LABEL(F15_POINTER)
    TRACE 'PATH = ', PATH(PLATFORM_PTR)
    MOVE PATH(PLATFORM_PTR)             TO F15_PATH(F15_POINTER)
    MOVE START_POSITION(PLATFORM_PTR)   TO INITIAL_POSITION(F15_P
    MOVE LAUNCH_TIME(PLATFORM_PTR)      TO F15_LAUNCH_TIME(F15_PO
    MOVE LAUNCH_POINT(PLATFORM_PTR)     TO F15_LAUNCH_POINT(F15_P

    MOVE INIT_LAT(F15_POINTER) TO INPUT_DATA THETA_IN
    MOVE INIT_LON(F15_POINTER) TO INPUT_DATA PHI_IN
    MOVE INIT_ALT(F15_POINTER) TO INPUT_DATA ALTITUDE_IN
    CALL ALTITUDE_TO_XYZ IN COORDINATE_CONVERSIONS GENERAL
        USING F15_ALT_XYZ_INTERFACE
```

For Help, pre | Ln 11, Col. 44, CW | DOS | Mod: 3/6/2014 5:29:09PM | File Size: 310

```
CONTROL SECTION
    TITLE, GLOBAL PLANNING TOOL
***    TRACE

LIBRARY SECTION
    C:/S/LIBS/GENERAL
    C:/S/LIBS/RTG_DRAW
    C:/S/LIBS/SAT_LIB
    OVERLAYS_3D
    C:/S/LIBS/FPPS_W_LIB

GRAPHICS SECTION
    ACTIVATE GRAPHICS PERSPECTIVE
    BACKGROUND_COLOR BLACK
    WORLD_SPACE  LOWER_LEFT  = (-60000, -60000, -60000),
                 UPPER_RIGHT = ( 60000,  60000,  60000)
    VIEW FROM 11396.94, +44292.18, 35529.45 TO 0, 0, 0

    ICON F15       =   AIRPLANE  SCALE 0.0124, 0.0124, 0.0124
    ICON F18       =   AIRPLANE  SCALE 0.0124, 0.0124, 0.0124
    ICON A10       =   AIRPLANE  SCALE 0.0124, 0.0124, 0.0124
    ICON SHIP      =   CRUISER   SCALE 0.0124, 0.0124, 0.0124

    LINE F15_PLAT_2_PLAT         COLOR BLUE STYLE 1  THICKNESS 2
    LINE F18_PLAT_2_PLAT         COLOR BLUE STYLE 1  THICKNESS 2
    LINE A10_PLAT_2_PLAT         COLOR BLUE STYLE 1  THICKNESS 2

    OVERLAY 1 = DRAW_COUNTRIES     IN COUNTRY_DRAW
                                   IN OVERLAYS_3D
                                   AT (0, 0, 0)
                                SCALE 0.0024, 0.0024, 0.0024
                                 MENU COUNTRY_BOUNDARIES
                                 *** COUNTRY_DRAW_CONTROL

    ENABLE INSERT EVENT
    ENABLE UPDATE EVENT
    ENABLE PICK EVENT

    RTG_EVENT_HANDLER = MAIN_EVENT_HANDLER

PARALLEL SECTION
    NUMBER_OF_PROCESSORS = 6
    IND_MODULE IND_MAIN            PROCESSOR 1
    IND_MODULE IND_SATELLITES      PROCESSOR 2
    IND_MODULE IND_F15_PLATFORMS   PROCESSOR 3  ***, TRACE 0, 999999
    IND_MODULE IND_F16_PLATFORMS   PROCESSOR 4
    IND_MODULE IND_F18_PLATFORMS   PROCESSOR 4
    IND_MODULE IND_F22_PLATFORMS   PROCESSOR 6
    IND_MODULE IND_A10_PLATFORMS   PROCESSOR 5
    IND_MODULE IND_UAV_PLATFORMS   PROCESSOR 5
    IND_MODULE IND_SHIP_PLATFORMS  PROCESSOR 6
    IND_MODULE IND_E3A_PLATFORMS   PROCESSOR 6, TRACE 0, 999999
    DELTA_TIME = 240   ***  240 Seconds = 4 MINUTES
```

For Help, pre | Ln 39, Col. 1, CW | DOS | Mod: 3/7/2014 9:23:35AM | File Size: 2103

Figure 1. VisiSoft engineering drawing - just double click on the icons to edit the code.

REFERENCES

1. Michael Rothschild, "Bionomics," Upside Magazine, February 1996.
2. Deming, W. Edwards, *Out of the Crisis,* MIT CASE, Cambridge, MA, 1992.
3. Standish Group International, "Chaos, Charting the Seas of Information Technology," The Standish Group International Inc., Report, Dennis, MA, 1995.
4. Business Week, "PROGNOSIS '95", McGraw-Hill, January 9, 1995, pp 72-80.
5. Groth, R., *Is the Software Industry's Productivity Declining?,* IEEE Software, Nov/Dec 2004.
6. Wirth, Nicklaus, *A Plea for Lean Software*, Computer, Feb, 1995.
7. Upside Magazine, "Musings on the Millenium," Feature Editorial, October 1994.
8. Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Oct 2000.
9. Strassmann, Paul, "From a Craft to an Industry," Ada Symposium, George Mason University, 1992.
10. Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Oct 2000.
11. Krishnadas, L.C., EE Times Article, Jan 17, 2008, www.eetimes.com/showArticle.
12. Merritt, R., Wintel will fund parallel software lab at Berkeley, EE Times On Line, Feb 13, 2008, www.eetimes.com/showArticle.jhtml?articleID=206503988.
13. Wakabayashi, Daisuke, Reuters Article, Seattle, 03/13/08.
14. Christensen, Clayton M., **The Innovator's Dilemma**, Harvard Business School Press, Cambridge, MA, 1997.
15. Kuhn, Thomas, **The Structure of Scientific Revolutions***,* The University of Chicago Press, Chicago, IL, 1970.
16. Parnas, D., "Education for Computer Professionals," IEEE Computer, January 1990, pp 17-22.
17. Sitner, Jerry, "Viewpoint - How Much Longer," Mainframe Journal, July 1990, pp 120.
18. Yourdon, E, **Decline & Fall of the American Programmer***,* Yourdon Press - Prentice Hall, Englewood Cliffs, NJ 1993.
19. Anselmo, Donald, *Why Software Productivity Has Not Improved*, Software Summit, Washington, D.C., May 2004.
20. van der Linden, P, **Expert C Programming - Deep C Secrets***,* SunSoft Press - Prentice Hall, Englewood Cliffs, NJ, 1994.
21. Cave, W.C., et.al, **Time is of the Essence: Software Engineering For Parallel Processors**, Visual Software International Technical Report, Oct 2007, Spring Lake, NJ.
22. Cave, W.C., et.al, **Software Theory For Parallel Processors**, Visual Software International May 2014, Spring Lake, NJ.