Software Factory: Recent Advances in Disruptive Technology

Colin James III, Principal Scientist C: 719.210.9534 / F: 970.593.1350
1613 Morning Dr, Loveland CO 80538-4410 psum@cec-services.com

Introduction

This is about how to build a factory for accounting arithmetic software. One previous software factory was built outside the USA about 20 years ago with limited success. A difference between a hardware and software factory shows the difficulty that plagues software development: hardware either works or does not work, but software may sometimes work even though it is broken. The challenge is to make use of existing products to assemble the software factory and thus avoid the reinvention of any software. The solution is to rely on proven methods of development, recent advances in database design as logic table technology [LTT], and reusable components. The result is a factory that produces software that is portable, scalable, usable, and maintainable. One benefit of the effort is reliable statistics that describe the process of development. The final product executes in real time and faster than other accounting software. Results are generalized to processing, inventory, and control and applied further to all types of scheduling and manufacturing.

Client Information

The target client is the Financial Management Service of the United States Department of Treasury. The project name is Standard General Ledger [SGL] which is an accounting arithmetic system based on double-entry book keeping. The Government invented SGL and requires its use by all Departments. SGL consists of about 150 accounting transactions and 100 accounts. A typical transaction contains 43 debits and 38 credits. SGL operates on platforms as Windows NT desktop, UNIX compatible midrange, and IBM compatible mainframe. SGL may reside within a wrapper for Enterprise Resource Planning [ERP]. SGL is usually implemented in procedural programming languages as COBOL, *C*, and in the non procedural Structured Query Language [SQL] by embedding it.

Government projects suffer from scope creep, and to that SGL is no exception. To overcome that condition, the customer was eager to adopt and use arbitrary software development methodologies. The danger is that all methods were not invented equally. In practice, this means dynamic methods where pieces are added, such as the IEEE theoretical standard, may not be as effective as a comprehensive method that is tailored by the deletion of pieces, as Mil-STD-498 and its ancestor DoD-STD-2167A/ 2168. What follows is that success with a method depends upon how it is chosen and how it is followed by the leadership of the project.

Challenge

The challenge is building a software factory to make industrial strength applications that become strategic business assets. The situation is that no such factory exists as commercial off the shelf software [COTS]. Benefits sought are software output that is portable, scalable, useable meaning reliable, and maintainable. The constraint is developing a mix of tools, methods, and design to meet requirements.

Strategy 1 builds a software factory from scratch as a prototype and ostensibly couched in the political title of feasibility study. Current developmental methods promise to do it, implying instant gratification in producing something without responsibility for potential side effects such as those arising from minimal testing. Current developmental tools offer enticing promises of quick success within the visions of individual vendors. Some constraints are that: the prototype becomes the delivered project rather than the throw away code that it really is; and extreme methods with in the latest tools may not pass the timely measure of best by test. Benefits are quick delivery of something under the guise of development and job security by exposure to new tool sets.

Strategy 2 reuses some generic form of the first known software factory. In the 1980's Nippon Telephone and Telegraph [NTT] was the second largest consumer of Ada compilers, after the US Department of Defense. Ada is a procedural language portable to many hundreds of computer platforms. Ada is noted for support of multitasking and reuse by separate program specification and implementation bodies. NTT set up a factory to produce portable software components for the telecom industry. The details of the factory and back end database were kept secret because at the time NTT deemed the factory to be a significantly competitive advantage. Some constraints are that: Ada is no longer taught in schools and not used in new efforts; and the relational database with source code originally in Ada and the public domain, named AdaSAGE from the Idaho National Laboratory, is not maintained by them but elsewhere in non portable Modula 2. Thus, no benefits exist.

Strategy 3 uses COTS as back end database and front end access. The database should be relational and SQL compatible to make use of that standardized commodity. The access tools to the database should be portable. Constraints are that front end access tools may have limited portability. For example to use web pages for database access, queries require embedding SQL in procedural languages which may also vary widely by vendor fiat. This implies an implementation path which is inherently not portable or maintainable. Hence benefits are that: a good database design theoretically avoids procedural processing; non embedded SQL in the form of static triggers is portable; scalable relational databases are supported by vendors such as IBM DB2 and ORACLE; direct database design implies simpler deliverables that are reliable and usable; and simpler deliverables imply software that is maintainable with less leadership, staff, and payroll. The problem domain when fully generalized and abstracted implies the obvious boundaries of the solution domain. That solution domain also necessarily excludes the temptation to mix and match combinations of disparate strategies.

Solution

The solution domain is in these sequential phases: COTS choice; design; and implementation. The COTS platform is to use relational databases from the major vendors. The design of the database is based on LTT and its algebra. A logic table contains logic switches which the SQL engine reads based on input of the users and returns the tasks to perform. This design coerces non procedural SQL to do procedural processing. The granular database access code implemented in SQL is not embedded in procedural languages. To achieve portability the SQL code is wholly contained within a static trigger of less than 50-lines of code. A solution for front end access is to adopt the most portable stand alone tool which is Lotus Approach. This decision avoids endorsement of platform specific, non portable tools such as ASP, C, C++, C#, Java, and .NET.

The finished product is named pSUM, pronounced "Sum", for the acronym of portable, scalable, useable, and maintainable. pSUM has a logic table that is fully configurable and programmable by the user. The logic switches support: double entry book keeping with debits and credits; triple entry book keeping known as Momentum Accounting with trebits; and generalized N-entry book keeping. pSUM contains logic switches to: perform complex scheduling; emit automatic output as input to other logic tables; allow infinite customization depending upon the initial user input. The logic table also indexes itself to make it self-modifying in real time and thus is a self-contained artificial intelligence unit.

Historically, object oriented (O-O) software renames variables as objects, declares constraints as assertions, and promises that objects will follow asserted contracts. O-O software still fails to be portable. Next comes aspect oriented (A-O) software which promises delivery of products to capture and mimic precisely the way humans perceive and act. In manufacturing, attempts at A-O programming are schedule and priority dispatch applications based on the rules of artificial intelligence. As a finite automation, A-O software by definition cannot capture the complexity of the human mind and thus continues to break its promise.

The logical successor to A-O software is now subject oriented (S-O) software. pSUM is a S-O software solution by addressing all subjects associated with requirements by the use of the logic switches of LTT. The S-O software approach differs from that of data driven software because data is not directly a part of the object domain but ultimately is a trivial subset of the subject domain. In S-O software there are no O-O contracts or A-O promises, just the subject fully encapsulated within a finished product which has business value as a strategic corporate asset.

Disruptive Technology (DT) is defined as an innovation that will potentially affect the way of doing business in the next 20 years. What follows is that pure S-O software is an instance of emerging DT. LTT is the S-O programming core of pSUM and produces software that is portable, scalable, useable, and maintainable. Therefore the pSUM factory is S-O software that qualifies as DT.

Results

pSUM meets the requirement of a factory to emit software that is portable, scalable, useable, and maintainable. pSUM is also fast in real time performance. The average transaction for SGL makes only 14 accesses five tables. On a desktop computer with 2.4 GHz Pentium and all tables on one hard disk, pSUM performs 136 such complex transactions per second or 8160 transactions per minute.

A side product of the pSUM effort is the Software Development Methodology [SDM] based on Mil-STD-498 and Business Object Notation. A recent advance from SDM is the statistic that the time consumed in collecting requirements is 10% of the total effort and thus accurately predicts the time to completion of the project. An advance is the delivery and acceptance of the user manual as the requirements document before any code is implemented. A discovery is how to map attributes, objects, classes, and clusters on a one to one basis directly into columns, rows, tables, and views with constraints.

In pSUM the static trigger of less than 50-lines of SQL code reduces into an abstract form of Petri Net that is identical to the model of the Kanban cell. What follows is that LTT is compatible mathematically with just in time [JIT], flexible manufacturing systems [FMS], and advanced planning and scheduling [APS].

LTT applies to ERP and software development as follows:

[G/A] No ERP ‣ LTT Kanban JIT FMS APS LTT ←ERP [D/S]

Where ERP or software development does not exist, to implement LTT from the general or abstract level [G/A] to the detailed or specific level [D/S] is top down from APS to Kanban. Where ERP or software development does exist, to implement LTT from the detailed or specific level [D/S] to the general or abstract level [G/A] is bottom up from Kanban to APS.