Reuse so Far: Phasing in a Revolution

James M. Neighbors

SADA, 3482 Wimbledon Way, Costa Mesa, CA 92626, USA
neighbrs@netcom.com

Abstract

This paper presents a lifecycle model that corresponds to a factor of ten improvement in software development productivity. It is argued that this level of productivity may be met today without aiding the general software crisis. Success and failure in software is not just a matter of improving software development productivity. Software flexibility for a changing world must also be maintained.

Key words and phrases*: domain analysis, transformational implementation, software architectures.*

1. Introduction

The panel mandate is to identify reuse technology successes and failures. In addition, our moderator [1] has challenged us to examine whether any reuse technology makes a revolutionary difference. To some degree I have already done this in my keynote address [3] but Biggerstaff brings up some interesting points.

2. What does a revolution look like?

For the sake of argument I claim that a factor of 10 improvement in the production of software is a revolution. This is a safe definition since it has been estimated that software productivity only improved about 3% to 8% per year [5] over a twenty year time frame that included the rise of Software Engineering, high-level languages, and on-line computing. I derived a short lifecycle model [3] using Boehm's data [2] for new development of a large project. The lifecycle phases and their percent of the total effort are:

- *Requirements* (req, 6%) provide the function, performance, and external constraints on the system.
- *Analysis* (ana, 8%) interrelates the data, function and external interfaces to ensure that the requirements are complete and understood.
- *Design* (des, 29%) casts the understanding of analysis into an architecture of computer process structures and details the function of members of the structures.
- *Implementation* (imp, 34%) constructs actual program code and tests individual codes.
- *Testing* (tst, 23%) checks compositions of codes and functional performance of the resulting system.

None of these phases is independent. As an example, integration testing is directly related to architectural design. Similarly, functional testing is directly related to the functions derived in requirements.

I propose to show what a lifecycle providing a factor of ten productivity increase would look like by phasing out current lifecycle phases. Reuse technology so far has done some of this by starting to phase out implementation. The phase out of implementation has lessened the unit testing chore. I have used these phase relationships to propose new lifecycle scenarios (A to E) with the effort in each phase reduced (Fig. 1).

	req	ana	des	imp	tst
now	100%	100%	100%	100%	100%
A	100%	100%	80%	20%	60%
B	100%	100%	40%	10%	35%
C	65%	85%	20%	5%	15%
D	50%	60%	10%	0%	10%
E	50%	50%	5%	0%	5%

Figure 1. Lifecycle scenario vs. pct. effort from current lifecycle

I apply these reductions to the estimated large project man-month (MM) data [2] which is shown as the "now" row in Figure 2. Scenario E, which is an improvement over the succession of scenarios A to D, shows a factor of ten improvement in productivity (Fig. 2). The number of man-months is reduced from 416 to 39. Scenario E is what a revolutionary lifecycle looks like. As can be seen in Figures 1 and 2 most of the effort has been taken out of design, implementation and testing.

Panel position paper on "Reuse so Far" at Third International Conference on Software Reusability, November 1-4 1994, Rio de Janeiro, Brazil. Proceedings to appear IEEE Press.

	req	ana	des	imp	tst	ttl MM
now	24	32	121	141	98	**416**
A	24	32	97	28	59	**240**
B	24	32	48	14	34	**152**
C	16	27	24	7	15	**89**
D	12	19	12	0	10	**53**
E	12	16	6	0	5	**39**

Figure 2. Lifecycle scenario vs. man-months required in each phase of lifecycle

Technologies that aid human developers in these phases could be judged as "not revolutionary" because in the revolution, scenario E, these phases are practically gone. This seems to support Biggerstaff's comment [1] that "OOP (object-oriented programming) is a small technological delta that is somewhat out of the mainstream of this revolution."

3. Can we get there from here?

Can we get from our "now" lifecycle to scenario E? The surprising answer is we can get there today. The odd part is that it will not make much of an impact on the general "software crisis." I would expect the Domain-Specific Software Architectures (DSSA) projects [4] to easily achieve scenario E using very domain-specific program generators. Outside of the particular problem domain the general software problem remains unchanged. We could build hundreds of individual DSSA's but ultimately we would get tired of maintaining the large-grain components and would desire to reuse common components. This puts us back into the long-term game of transformational implementation.

4. Let's buy some parts

An alternative path to scenario E is to use large-grain commercially available parts, such as Microsoft Visual Basic VBX controls. The problem with this approach is that it combines the developing system version / configuration space [3] with those of all the parts you use. In the short-term this is a fine approach. In the long-term the version / configuration space may prohibit system implementation. As an example, you use VBX controls under Windows v3.1 to build your system and under Windows v9.6 all of a sudden VBXs don't work. The company that supplied your VBXs has, of course, gone out of business.

5. Flexibility: either bend or break

Change is constant. As software experts we surly know that our systems requirements, development techniques, tools and hardware all change at a dramatic rate. Many of the concepts I advocate [3] (e.g., structural architecture, version space, configuration space, and specialization) are only for flexibility. The resulting system embodies these concepts but they do not change the system's function. OOP has been successful because it aids flexibility. From the discussion in the previous sections we can infer one thing: flexibility costs more.

6. Success and failure

The success or failure of a technology depends on the goals to be reached. If the goal is to create a lot of systems in a constrained problem area, then the individual DSSA's should be extremely successful. If the goal is to solve the software crisis in general then the DSSA's will be a failure except with respect to how they contribute to our knowledge of how to solve the general problem.

Similarly, if the goal is to create short-term solution systems that don't have to be maintained over time, then the use of techniques like VBX is a success. However I would expect building a telephone system or global network out of these to be a big failure.

Work on the general software crisis currently languishes around scenario A waiting for a lot of work in transformational implementation. It's a failure on all counts. Occasionally some nice things like domain analysis do come out. I like this area because if we do get anywhere it promises to significantly change what people can do with computers. We will need the flexibility.

References

[1] Biggerstaff, T., Is 'Technology' a Second Order Term in Reuse's Success Equation?, **Proceedings 3rd Intl. Conf. on Reuse**, IEEE Press, to appear 1994.

[2] Boehm, B.W., **Software Engineering Economics**, pp. 66, Prentice-Hall, 1981.

[3] Neighbors, J.M., An Assessment of Reuse Technology after Ten Years, **Proceedings 3rd Intl. Conf. on Reuse**, IEEE Press, to appear 1994.

[4] Mettala, E., The Domain-Specific Software Architecture Program, Special Report CMU/SEI-92-SR-9, CMU Software Engineering Institute, June 1992.

[5] Morrissey, J.H., and Wu, L.S.-Y., Software Engineering ... An Economic Perspective, **Proc. 4th Intl. Conf. Sfw. Eng.**, pp. 412-422, IEEE Press, 1979.