It Was Winston, Not Walker, Royce

In the May/June 2003 From the Editor message, you referred to a paper by Walker Royce on the waterfall lifecycle model published in 1970 (along with a footnote identifying the paper). This paper was written by Winston W. Royce, not Walker, who I believe is Winston’s son.

And there was no reference to “waterfall” anywhere in this paper other than graphically. Most of his diagrams have a cascading structure to them, which no doubt led many readers to use the term waterfall in reference to his model. However, he never used the term to refer to his model.

David Ramger
Veritas Software
david.ramger@veritas.com

Pick the approach that fits

Just to put my two cents (probably less) in, I do not think that any of the classical approaches are “over the hill,” even though they might have had their creation in the paper tape/card era of computing. I believe that the successful application of most formalized approaches is situational, in that different approaches apply to different problems, requirements, and constraints. Also, the existence of “useful” approaches that are old (at least in software terms) helps provide some of the engineering basis for our profession.

On another topic, in the May/June Loyal opposition column: While I have Pete McBreen’s recent book on XP, I have been involved in numerous programming projects using a large variety of software development approaches, from waterfall to spiral to OO to XP. I most heartily support McBreen’s contention that “software methodologies are situational.” Indeed, you should look at your project’s end goal, the typical constraints (schedule, function, cost, personnel, and so on), and apply a measurable methodology. Note, critical measurement is a must to avoid some of the most common failures of software projects. The only way to do this would be to have a critical look at the suite of approaches used in software development (I’m not sure if any are really conceptually obsolete) and apply the most appropriate one. This requires good critical thinking, which it looks like Pete has done for XP.

Thanks for bringing this book to my attention.

Paul W. Horstmann
Director, NIIP Virtual Enterprise Development IBM Software Group—Strategy and Solutions
horstman@us.ibm.com

As Donald Reifer points out in his May/June Manager column, XP and CMM are philosophically—and in my opinion otherwise—compatible. Both focus on improving...
the predictability of producing a software artifact. CMM takes an actuarial approach and tries to reduce the variation among projects in the difference between predicted and measured quantities—that is, schedule, effort, and so on. XP takes a clinical approach and tries to find acceptably small yet realistic values for these quantities in a given project.

Of course, CMM also addresses a wider range of software development issues than XP, not just issues such as project tracking and oversight or quantitative process management—the key process areas that seem particularly onerous to XP teams. CMM also addresses issues such as subcontractor management and configuration management. These are largely independent of an organization’s development method and might well dominate the predictions for a given project.

Thus, XP and CMM can complement each other well. Problems such as the two described in Reifer’s article arise when we lose sight of our goals and participate in a dogmatic “my method is better than your method” dispute so typical of our software community.

Dan Kalcevic
Principal consultant
T-Systems GEI GmbH
daniel.kalcevic@t-systems.com

What you don’t measure

I fully support the title of Nancy Eickelmann’s and Animesh Anant’s Quality Time column in the March/April issue: “Statistical Process Control: What You Don’t Measure Can’t Hurt You.” I would even extend it by saying “What you do measure could prevent you from being hurt.”

At Ericsson R&D Netherlands, we initially measured fault density of various test phases. But each time the fault density was above the upper control limit, we had the same discussion: Did we have a high defect density because this product has poor quality or because we did a thorough testing job? When fault density was below the lower control limit, we asked: Was this due to a high-quality product or poor testing?

Similarly to what you conclude in your article, we simply needed some additional measurements to get good insight. We now measure two things: How many defects are made in a certain design phase? And how many defects did we find in a certain phase?

After release, we measure the defects that customers have reported in
the first six months. After that, we do normal defect tracking as part of our maintenance support.

By modeling these measurements on the project phases, we have gained much more insight on the quality of our development and verification phases. We have been able to make better decisions regarding the quality of our design processes and the effectiveness of inspections and testing. We also have been able to estimate the number of latent defects in the delivered product and to release our product earlier than scheduled, with a known quality.

Our project defect model supports your Scenario 4: We are taking figures of the initial number of defects into account to measure defect detection effectiveness and release quality. We have extended this with verification effectiveness to estimate product quality. On the next project, we will define targets for inserted and detected defects so that we can estimate release quality.

Of course, estimating the number of inserted defects is not easy. But even rough estimates have helped us to get a good discussion going on product and process quality, far beyond poor versus good based on fault density. So, our organization has certainly benefited from measuring more than just fault density.

Ben Linders  
Operational Development and Quality  
Service Network and Applications  
Ericsson Telecommunicatie B.V.  
ben.linders@etm.ericsson.se

A component-oriented process model

It was a nice coincidence to read the article by Ali H. Dogru and Murat M. Tanik entitled “A Process Model for Component-Oriented Software Engineering” (March/April 2003), because we have been studying the same subject for a long time. Component-based software development can now solve some classical problems of software development and help make software engineering a real discipline similar to its traditional counterparts. Although CBSD’s promise is good, there are some major obstacles ahead.

As Dogru and Tanik stated, we need to define a methodology, similar to UML, that would guide CBSD teams from requirements identification to design, implementation, testing, and even maintenance. The authors’ component-oriented software engineering approach, COSE, might fulfill such a need, but we must carefully review it and identify possible shortcomings.

First, component-based methodologies must clearly state how they unify or differentiate the two types of CBSD styles: component production and component-based system construction. The COSE methodology does not directly touch this issue. My work with colleagues shows we can integrate both development styles into a unified methodology that represents component-based software systems as an architecture of recursively decomposed components. At each level of decomposition, we treat components as new, stand-alone software systems, paying attention only to the special integration requirements at that level. So, both system developers and component producers would use the same unified methodology.

Another problem in COSE is that it does not address component-based architectural issues. Software architectures are used to identify how the overall system would be constructed out of building blocks (here the building blocks are the components) and evaluated to satisfy requirements. CBSD methodologies must be architecture-centric, providing iterative, incremental, reuse-driven processes to be applied. Because COSE does not address component-based architecture issues, it is not easy to evaluate the success of a system constructed according to this approach.

COSE also neglects the specification of nonfunctional characteristics in CBSD. Any methodology in the field must provide techniques to guarantee that component-based software products will satisfy a measurable level of reusability and quality. The COSE approach must be able to identify and satisfy software’s nonfunctional characteristics.

Finally, COSE’s approach to the inheritance of components is not clear. It seems to employ inheritance in its component production, which we think is unadvisable.

As the authors state, their research is not yet complete. COSE might eventually provide a clean and mature view of component-based software production.

Yusuf Altunel  
Instructor, Istanbul Kultur University  
PhD student, Eastern Mediterranean University  
y.altunel@iku.edu.tr

**CBSD methodologies must be architecture-centric, providing iterative, incremental, reuse-driven processes to be applied.**
Ali H. Dogru and Murat M. Tanik respond:

Your point about style coverage is well taken. Our approach definitely aims at component-based system construction. What we promote is actually beyond component-based approaches. We only leverage such technologies to define the “build by integration” paradigm, which excludes and even tries to avoid code development of any kind (including components). This is the long-overlooked power hidden in the component concept. Component production is not a fundamentally new problem, but system construction without code development is. That is why COSE is committed to system construction.

Our opinion about a unified methodology accommodating both styles is very different. A unification attempt would restrict the basic component definition of third-party integration and separation of concerns. The nature of different domains might also require different methodologies for producing components. But we are investigating just the integration paradigm; conceptually, we treat components as components regardless of their domains. Domain orientation is a natural extension of our topic that deserves study.

Your comments about architectural issues are correct in the context of methodologies, but our approach is a process model, not a full-fledged methodology (as our terminology section clarifies). It tries to remain architecture independent. For the same reason, we did not mention the specification of non-functional characteristics. Eventually, our paradigm must be supported by tools.

The hottest discussion issue about COSE probably will be on its usage of inheritance. We are sorry the article’s limited space did not let us explain this further. If read carefully, the article does not suggest usage or non-usage of inheritance in component construction. Component construction is not of concern. To aid model understandability when two constructs are similar, we use the inheritance link to logically relate abstract components to each another. Components, on the other hand, should be obtainable from different parties, so using inheritance in their construction loses meaning.

It is not difficult to see how we differ. The article promotes a paradigm shift, whereas the letter anticipates more methodology-level techniques. Your ideas, however, are worth another article: a unified methodology will definitely prove useful.

Final Call for Papers and Call for Participation

28th NASA/IEEE Software Engineering Workshop

Greenbelt Marriott Hotel, Greenbelt, MD (Washington DC metro)

2-4 December 2003

Submit 4-page electronic abstract and queries to sew28@listserv.gsfc.nasa.gov by July 30th

Further Details available via: http://sel.gsfc.nasa.gov

Registration now open at: https://newton.computer.org/conferences/sew03.nsf/

ALL WELCOME!!

Sponsored by:
NASA Goddard Space Flight Center, Software Engineering Laboratory
IEEE Computer Society Technical Council on Software Engineering