

Is This a Revolutionary Idea, or Not?

Assembling the individual pieces to form an integrated component architecture.

Many people claim to have developed a scheme to “revolutionize” software development. In fact, the plague on the house of software is, and has been for years, hype. From tools to techniques to methodologies, gurus have claimed that whatever barrow they happen to be pushing at the moment is some kind of universal elixir, yet another magic bullet to cure all the supposed ills of the software field.

I heard another such collection of claims at a presentation several months ago. This time, however, things seemed a bit different. The person making this particular presentation was not only an academic with substantial qualifications but, more to the point, a consultant who had worked with significant industrial-strength projects. And his ideas jibed with my practitioner-formed preconceptions. Was this “revolutionary” proposal, I found myself wondering, different?

With that in mind, I’d like to ask for your help. Let me describe

this new idea for you, and you tell me whether it has promise or not. My initial screening is that it might. However, not having applied it in practice, I am not sure.

Here’s the story. I first heard the idea presented by Professor R. Geoff Dromey, of Griffith University’s Software Quality Institute, in Brisbane, Australia, at an academic seminar series in Brisbane. The idea is about constructing a software system from its require-

ments. Nothing exciting there yet, right?

But here’s what’s supposedly different about it. Each requirement, Dromey said, should be translated into its corresponding “behavior tree” (which describes the behaviors that will result from that particular requirement). A behavior tree, Dromey says, is made up of components (the software pieces), states (that those components can take on), events and decisions/constraints (that are associated with the components), data that the components exchange, and the causal, logical, and temporal dependencies (associated with component interactions).

It is the way those behavior trees are put together that makes things get interesting. To integrate those behavior trees, Dromey suggests they be placed together like a jigsaw puzzle, where clear points of intersection between the trees cause the puzzle pieces to fit together (Dromey claims that those points of intersection are

Complexity can be handled piecewise via integrating the localized behavior trees, rather than as one big global cognitively daunting task. This greatly reduces the strain on our short-term memory.

surprisingly easy to identify: two requirements integrate when one requirement in the course of its behavior establishes a precondition that the other needs for its behavior to happen. All requirements of a system have such a precondition). The beauty of this approach, he says, is that these puzzles can be assembled by proper placement of the pieces into the whole, a task that is to a large extent independent of the order of their placement. Using this approach, a software system can be built “out of its requirements” rather than just “satisfying its requirements.”

Following the translation and integration of the requirements, Dromey says, it is time to evolve the requirements representations into design/architecture representations. This design activity not only creates a solution- (as opposed to a problem-) focused representation, but it also tends to rationalize the components from which the completed system will be constructed.

Dromey calls the first phase of this architectural process the “Component Architecture Transformation,” and the output of that phase is a “Component

Interaction Network.” The primary thing that happens here is that components, which may be represented at many points in the requirements representation, are isolated out to appear only once in the solution representation. This process, begins at the root of the integrated requirements tree and moves systematically down that tree toward its leaf nodes including each component and each interaction that has not been previously included. This amounts to algorithmically transforming the integrated tree of requirements into a network of components that interact (the traditional conceptual view of a system).

Dromey calls the final stage of this process “Component Behavior Projection.” Here, component behaviors are concentrated by separately projecting each component’s behavior from the integrated requirements tree. The result of this process is a skeleton behavior tree for each component that will deliver the behavior it needs to exhibit to function as an encapsulated component in the component interaction network.

Dromey likes to refer to this whole process as “Genetic Soft-

ware Engineering,” in that the behavior and component representations, in a sense, are similar conceptually to the genes and DNA of a biological process. I am less than enthusiastic about this alternate name because the term “genetic” tends to be used in another context in software engineering, one which I happen to believe is of dubious validity. But the naming of the approach is beside the point; the question here is, is this a workable and advantageous approach to software construction?

The whole point of this approach, Dromey says, is to master the complexity that accompanies building a significant software system. Complexity can be handled piecewise via integrating the localized behavior trees, rather than as one big global cognitively daunting task. This greatly reduces the strain on our short-term memory.

A benefit of doing integration is that just as a picture emerges when all the pieces of a jigsaw puzzle are put in their correct places, a similar thing happens as the behavior trees of functional requirements are integrated. Surprisingly, the picture in this case is the inte-

grated component architecture of the system, along with the integrated behavior of each of the components in the system.

Dromey claims requirements integration has the additional benefit of being a powerful way to find defects in a system early—only when a requirement is seen in the context where it is applied do we see its problems.

As a fringe benefit of this approach, the traceability of the original requirements into the as-built software system becomes much easier. Even with the use of commercially available tools, it is well known that requirements traceability is a complex and barely manageable task due to the “requirements explosion” caused when the original requirements explode into the requirements for a design to satisfy those requirements. (Some researchers have found that explosion rate to be on the order of 50:1.) With Dromey’s approach, adding a new requirement is like adding a new piece to a jigsaw puzzle that was originally incomplete.

There you have it. A fairly simple idea, accompanied by a kind of satisfying metaphor (software requirements integration as the outcome of a jigsaw-puzzle-piece process). I will ask again: is this a revolutionary idea? Or is it, as too many other claimed software revolutions are, yet another variation on an old theme (or worse yet, a truly dumb idea masquerading in colorful marketing clothing)?

I would like to receive your input on this topic. This is one of those ideas, I think, that must be tried in practice on a significant project to see if it has the merit its originator claimed for it. Success with the method, Dromey claims, depends on doing requirements translation rigorously. If you try it out, no matter the result, I’d like to hear from you about your experiences using this method.

End Note

This column is based on several sources of input: R. Geoff Dromey’s presentation at a seminar series at the Queensland University of Technology; his article “Genes, Jigsaw Puzzles, and Software Engineering” and “From Requirements to Design: Formalizing the Key Steps,” (Invited Keynote Address, SEFM-2003, IEEE International Conference on Software Engineering and Formal Methods, Brisbane, Sept. 2003); and his evaluation of this column to ensure I’ve accurately portrayed his ideas here. Several papers, published references, tools, and other information describing Dromey’s approach are available at www.sqi.gu.edu.au/gse/. 

ROBERT L. GLASS (rlglass@acm.org) is editor emeritus of Elsevier’s *Journal of Systems and Software*, and the publisher/editor of *The Software Practitioner* newsletter.

© 2004 ACM 0001-0782/04/1100 \$5.00

2005

EDITORIAL CALENDAR FOR COMMUNICATIONS OF THE ACM

January	Interaction Design and Children
February	Medical Modeling
March	The Disappearing Computer
April	Transforming China
May	Distributed Auto-Adaptive and Reconfigurable Systems
June	3D Hard Copy
July	Designing for Mobility
August	Spyware
September	RFID Technologies and Issues
October	Technology and Societal Influences
November	The State of Computer Security
December	The Semantic E-Business Vision