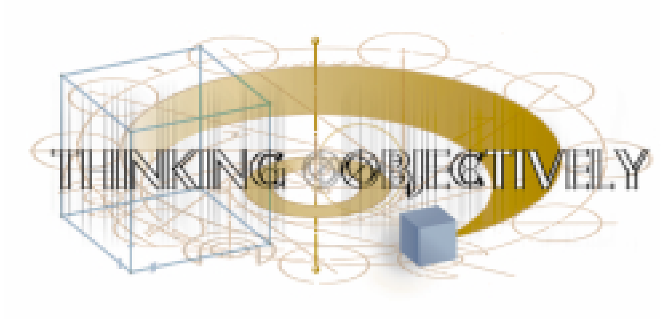

Mauri Laitinen, Mohamed E. Fayad, and Robert P. Ward

The Problem with Scalability

In software, very few projects remain of constant magnitude.

Software engineering evolved to deal with the needs of large systems developed by multiple large teams, and it has struggled to keep up with the increasing size and complexity of large systems ever since. Meanwhile, the typical small system has become larger and the typical team size has, if anything, gotten smaller. This situation leaves a mismatch between the methods developed for large projects and the smaller but significant projects that make up much of the software development industry.

Unlike traditional engineering, the targets of software engineering continually become larger. Contrast this with civil engineering. While we can build much larger structures today, a large class of structures remains constant. A bridge built over a creek 500 years ago is the same size as one built this year. What has changed are the methods and materials used to build the bridge. In software, very few projects remain of constant magnitude, so it is difficult to directly compare the methods and tools over time.

This growth has led to a few consequences. First, the small project of today is substantially larger than the small project of even 10 years ago. And second, we have overlooked the downward-scalable range of software engineering tools and management methods.

Using Battleship Blueprints to Make a Dinghy

Large systems, too complex for individual comprehension, must be subdivided into smaller tasks coordinated between groups. In fact, a large portion of software engineering is devoted to the documentation, notification, and management review needed to coordinate large projects. Attempting to scale down is more problematical. Software engineering texts, if they mention smaller applications at all, usually recommend that existing techniques be scaled down to fit the resources of the organization. This approach assumes that scaling down is just the compression or elimination of some methods or development phases.

A small group is likely to reject the idea of using large-scale methods, arguing that it makes no more sense than to attempt scaling down battleship blueprints to build a dinghy. And since much of the small project data is old enough to not reflect current development environments and methods, we have little in the way of examples or guidelines. This lack has led to some "less than useful" recommendations.

- ***Just scale it down***. This advice sounds good until put into practice. If much of the software process is devoted to managing multiple groups rather than dealing with some unique characteristic of software, then identifying what to keep and what to drop is not obvious. If you don't drop any of the methods, then how, exactly, do you go about scaling them down? This leads to the next questionable recommendation.
- ***Developers in small groups must play many roles***. This could also be called "The Lucy Syndrome." The 1950s television show, "I Love Lucy," typically created situations where the star, trying to keep up some illusion, ran frantically about in a series of disguises. As things became more frenetic, her roles became increasingly confused, ultimately collapsing the whole scheme. Piling on the multiple roles required of a large organization on a small group is simi-

If you don't drop any of the methods, then how, exactly, do you go about scaling them down?

larly nonsensical and as doomed as "Lucy." Clearly, members of a small development group play many roles, but these are not necessarily the same ones needed for a larger organization. The difficulty instead, is to decide what roles are important for developing good software and what roles are important for coordinating and managing multiple groups.

Scaling down cannot mean that we should keep the same tasks and somehow make them smaller, nor can it mean that we should have fewer people doing more. At a certain point, a single person will have much more to do than he or she can possibly do accurately. In fact, a scaled-down group must have exceptionally talented and motivated people in order to function. In large groups, having irreplaceable people is a weakness; in small groups it is a necessity.

Taking this discussion further, there are a number of problems with scaling down software engineering methods. Here we list a few:

- So much of the software engineering literature is designed to separate functions in a way that may have little meaning or value in small organizations. In fact, it is the synergy of people with diverse skills working together (without management intermediation) that gives the small group its edge.
- Some methods are not apparently scalable. The largest segment of this class is management and reporting. A formal report to oneself or from a small group to a small group doesn't make a lot of sense. While PERT charts and activity matrices are extremely valuable on large projects, their overhead becomes harder to justify as the work is scaled down.
- Fixed method costs (or management costs) tend to become less sustainable as a project gets smaller. For example, if it takes 30 minutes, including production of management reports, to track a task's progress, each 100 tasks takes 50 hours. On a large project this overhead might be acceptable and even necessary, but it could well be too high for a small project.
- Tool overhead is another aspect of fixed costs. Both the cost and learning curves of development tools, especially CASE tools, can be harder to justify in smaller projects. Note that a significant portion of software development is performed by startup companies that cannot bank on amortizing the costs over multiple projects.

If our models are best for large groups, does it mean we should increase the size of smaller groups? No. There is evidence that smaller groups are more efficient. Smaller groups usually have higher per-capita productivity than larger groups. Moreover, while many software engineering methods have proved valuable, the totality of these methods has not been proved optimal for small groups. We must start studying the development needs of smaller groups and develop methods that work for them.

But What is Scalability?

We use the term "scalability" almost without thinking. When we talk about scalability, we think of software such as Unix, that can scale from PCs to large servers. Scalability is a fundamental quality of software. The same operations can be used without wear on programs of all sizes and with any volume of data. But scalability is not a limitless quality. In order to make the term meaningful, it has to be understood within a particular context and then regarded as variation within a range. So first, we should develop an understanding of the definition of scalability.

Scalability in the context of software engineering is the property of reducing or increasing the scope of methods, processes, and management according to the problem size. One way of assessing scalability is with the notion of scalable adequacy—the effectiveness of a software engineering notation or process when used on differently sized problems. Inherent in this idea is that software engineering techniques should provide good mechanisms for partitioning, composition, and visibility control. It includes the ability to scale the notation to particular problem needs, contractual requirements, or even to budgetary and business goals and objectives. Methods that omit unneeded notations and techniques without destroying overall functionality possess scalable adequacy.

Tailorability—another way of assessing scalability—is the customizability of a technique or a process to a specific domain or of standards. If process and notational changes can effectively be made, then a system is tailorable. For example, object-oriented techniques, such as UML, can be tailored to fit different problem domains and organizations. A process, such as configuration management is usually adapted to the development project. And development standards like DoD-STD-498 must be altered to fit a particular organization, contact, and problem domain.

Unfortunately, there is no existing process for scaling up or scaling down that addresses large changes in problem size. In most cases a major change in scale results in a fundamentally different method or different process. We are more familiar with scaling up, and what is similar to other engineering disciplines, but we don't know much about scaling down. In particular, scalable adequacy and tailorability must look at overhead and learning curves as notations are scaled down. Here are a few examples:

- Decomposition is a special case of scalability that is concerned with breaking a problem into manageable components. Decomposition is a basic technique of many development methods and is especially useful. However, as the development group gets smaller, coordinating tasks can become more difficult, especially without the prohibitive overhead of expensive CASE tools or support people performing task coordination.
- The presentation of multiple views is a form of notational scaling. A system under development usually interacts with different stakeholders in different ways. Customers, interested in the overall design, do not want the distractions of low-level details, while various developers will need to see detailed designs. In general, supporting multiple views of domains and multiple layers of complexity inside a domain is difficult and requires sophisticated CASE tools. The cost and learning curves of such tools often put them out of the reach of small groups.
- Large projects are usually developed under contract and must often conform not only to specified development standards but to organizational standards as well. In many cases, conformance to CMM or ISO 9000 organizational standards is as important as fulfilling technical project requirements. The same conditions hold for small companies performing certain contract work. However, for companies developing prepackaged software, Internet software, and in-house commercial products, standards conformance is less important. Customers are not especially interested in the product's conformance to various software development standards. Rather, they care about cost, utility, and interoperability with other existing software. Organizational and developmental standards may be important internally if they provide a clear competitive advantage, but they are not perceived by the customer as adding value.

For small software development groups, this presents another aspect of the scalability challenge. Without contractual requirements and without any marketing advantage in standards usage, all that remains is to determine the value of the standards in terms of development improvement. Clearly, if the standard cannot be economically tailored to the development, it cannot and should not be used.

In summary, scaling down is not a simple task, and it has clear limits. Methods requiring too much overhead for their relative benefits are ultimately not sustainable. This is the central problem with scaling down large, formal-communication-laden systems. A method is scalable only if it can be applied to problems of different sizes without fundamentally changing the method, and it is entirely unclear that many methods can be scaled down without such change. Moreover, scaling forces significant changes in the software architecture, software processes, methods, life cycles, and domain knowledge that usually introduce new sets of errors. This by itself puts a limit on scaling. Because we have been so focused on large-scale development, software engineering methods have tended to intertwine management and coordination with technical aspects. Deciding how to extricate these two areas and deciding how to make some methods work in small organizations will take some study. C

Mauri Laitinen (mdl@sierra.net) is a Principal in Laitinen Consulting at Lake Tahoe, CA.
Mohamed Fayad (fayad@cse.unl.edu) is J.D. Edwards Professor at the University of Nebraska, Lincoln.
Robert P. Ward (robert.ward@moai.com) is a software manager at Moai in San Francisco, CA.