Intro to Software Methodologies

Alex Sverdlov
alex@theparticle.com

1 Introduction

Here is a meta-algorithm to solve all problems: *If a problem is obvious and can be solved quickly, then solve it, otherwise break the problem into smaller pieces, and repeat.* Software Methodologies is essentially a study of various ways of applying the *algorithm-to-solve-all-problems* to the task of building software.

This general divide and conquer approach is used everywhere, in many industries. In fact, many popular algorithms are divide-and-conquer style (e.g. quick-sort, merge-sort, etc.).

Dividing helps in reducing the scope of each unit-of-work, as well as allow for parallelism, as many unrelated tasks can be performed in parallel by multiple developers.

That said, in order to divide a problem into pieces, we often need to have some understanding of the bigger problem and the resulting pieces. Such understanding may not be readily available at the time the decisions need to be made (suppose requirements are vague, or it is not clear how to break the task into smaller pieces).

There are many tasks that we do every-day and that we can train machines to do very well (such as recognizing images, faces, voice, etc.), but we just can't verbally explain how to subdivide into smaller tasks (we know we can train a neural network, but how exactly it handles the task, or how reliable it is, is still a mystery)

At some granularity level, the divide-and-conquer should stop—and it is not always clear what the granularity level should be. For example, if a task can be performed by a developer in 2 weeks, should it be broken down into 1-day sub-tasks? Perhaps 1-hour mini-sub-tasks?

The other item is integration. Once the sub-problems are solved, how are they integrated—what are the dependencies (do some problems need to be solved before others).

2 Why?

When it comes to building software, the question that is often asked is: why do we need a process in the first place? The simpler answer: repeatability. Even if a no-process approach worked on a past project, there is no systematic way of replicating that success in subsequent projects.

Even the 'no-process' is often mis-branded 'some-lightweight-process'; there are very few truly no-process development teams.

We need some way of cutting the big problem into smaller chunks that can be solved and later integrated—this meta-algorithm always exists, no matter what folks call it.