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. To build a house, we partition the task into building foundation, framing, electrical, plumbing, etc. We do something equivalent in technology all the time: 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, allows for parallelism (unrelated tasks can be performed in parallel by multiple developers), and allows for specialization (some tasks may require specialized skills).

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).

Partitioning the problem also highlights dependencies: different units-of-work may need to be done in a particular sequence and by different teams. The other end of the process is integration: gluing the completed units-of-work into a working-whole is often non-trivial.

Then there are tasks themselves, which may have a non-trivial technical complexity that cannot be split away: 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)

Errors are easier to correct when they are caught early—so it is vital to ensure that bugs are spotted/corrected early in the process, hopefully contained within a single module/component. Bugs discovered during integration are often harder to identify and subsequently cost more to correct.

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?

2 The What Question

Building software is a creative process. Every software project is unique, and has never existed before. The challenges are also often unique, and show up in unique ways.

It is important to know what is being built. It is kind of obvious, but what-is-being-constructed is often not well defined, or is confused with tools or implementation details.

This *what* question is often addressed in user requirements, or business level documentation.

3 The How Question

Once we know what we are building, the *how* question turns up. This is often addressed in a technical specification or analysis document. While the What and How descriptions may overlap, it is a good idea to conceptually keep them separate.

4 Why study Software Methodologies?

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.