Software Development Lifecycle

Alex Sverdlov
alex@theparticle.com

1 Introduction

Building software is essentially a structured way of solving problems—we often solve problems by building one solution module at a time, and then combining or composing them together into the big problem solution.

The same can be said about the software building process itself. We can decompose the process into distinct phases, which helps in planning and scheduling, as well as focusing the specific skills needed at each phase.

Historically folks have tried lots of different methods—and each has worked with some success in a certain setting. That said, there is no single-fool-proof way of building large systems. The methodology of the software development process does not guarantee success, but may contribute to failure.

The biggest hurdle with all methodologies is that creativity and success cannot be planned—it just happens. Often a complicated task turns out to be simple due to developer's insight—and often a trivial task turns into a complicated nightmare.

2 Prototyping

One under-represented software-development phase or method is *prototyping*. Prototypes are not meant to be solutions—just a quick and simple solution to a tiny subset of the big problem.

Often such prototypes highlight big problems (or big opportunities), that would not be obvious under other circumstances. So big suggestion: whatever you are doing, start with tiny prototypes to ensure that the core pieces of ideas are sound.

3 SDLC

The traditional software development lifecycle (SDLC) defines phases or steps that the development process goes through. These steps allow the immense task to be partitioned into chunks that can be scheduled, and approached differently.

The usual steps are:

1. Requirements gathering
2. Planning
3. Design
4. Coding
5. Testing
6. Deployment
7. Maintenance

3.1 Requirements

Every software is built for someone in mind. In the system/software requirements gathering, we interact with the intended users or customers of the software to understand their needs, and document the requirements.

The output of this process is documented requirements, often in the form of a business requirements specification (BRS).

At the minimum, this artifact must document who will use the product, and what the primary expected outputs.

3.2 Planning

Once the requirements are documented, we need to determine whether they can be developed in the allocated budget and time.

There is now a bit more technical details added to the business requirements specification, creating a software requirements specification (SRS).

This phase often limits scope of the project.

3.3 Design

The design phase continues on the planning phase, by documenting software and hardware design aspects for the system.

What environment is to be used: web-based app vs native desktop app.

3.4 Coding

This is the implementation phase of the design. This is often the longest phase of the process—and the phase where most of the problems show up—often problems that were introduced during planning or design phases.

3.5 Testing

There is a need to ensure that the produced code matches the design document—and that the requirements are satisfied.

Testing can take on many forms, such as automated unit tests, or spot checks, etc.

3.6 Deployment

Once the product is built, it needs to be deployed for customer use. This may involve installing several components to ensure proper environment, such as database deployment, sever deployment, etc.

3.7 Maintenance

After the development and deployment, the lifecycle continues. The software requires maintenance: corrections of bugs that show up in production and updates to cover more business use cases.

4 Agile

The popular "new" methodology is Agile. Lots of new methodologies describe themselves as "agile". The basic idea of agile is to be: ...agile regarding development, and to tighten the loop between requirements, development and feedback from users.

To ensure rapid development, there is a need to minimize errors during coding—so unit testing is often used to verify that coding changes in one part of the system do not break other parts of the system.

Once code changes can be made reliably, it becomes feasible to quickly prototype solutions and have confidence that nothing major breaks.

With quick prototypes and changes, users can be looped in for quick feedback. Quick feedback identifies requirements and design problems early, making them easier to fix (by using more quick reliable changes), etc.

Another aspect is 'pair programming', where two developers are sitting shoulder-to-shoulder working on the same piece of code. This actually works well in many settings—as the ideas bounce around and code gets developed quicker.

Also, with quick round-trips between "changes" vs "feedback", it becomes feasible to build a project schedule around such few-week sprints (as opposed to months of development without feedback from the users).

There are quite a few issues with Agile that most Agile proponents don't like to mention: some creative steps often can't be decomposed into short-two-week cycles.

Often the task driven nature of sprints limits the ability of going off-rails and trying arbitrary things to see if they'd work better. The ultra-scheduled nature of projects can often choke off creativity.