Every program has a life cycle. It doesn’t matter how large or small the program is, or how many people are working on the project – all programs go through the same steps:

1. Conception
2. Requirements gathering/exploration/modeling
3. Design
4. Coding and debugging
5. Testing
6. Release
7. Maintenance/software evolution
8. Retirement

One’s program may compress some of these steps, or combine two or more steps into a single piece of work, but all programs go through all steps.

Although every program has a life cycle, there are many different process variations that encompass these steps. Every life cycle model, however, is a variation on two fundamental types. In the first type, the project team will generally do a complete life cycle – at least steps 2 through 7 – before they go back and start on the next version of the product. In the second type, which is more prevalent these days, the project team will generally do a partial life cycle – usually steps 3 through 5 – and iterate through those steps several times before proceeding to the release step.
These days the management of software development projects generally fall into two different types, traditional *plan-driven models*,¹ and the newer *agile development* models.² In the plan-driven models, the process tends to be stricter in terms of process steps and when releases happen. Plan-driven models have more clearly defined phases, and more requirements for sign-off on completion of a phase before moving on to the next phase. Plan-driven models require more documentation of each phase and verification of completion of each work product. These tend to work well for government contracts for new software with well-defined deliverables. The agile models are inherently incremental, and make the assumption that small, frequent releases produce a more robust product than larger, less frequent ones. Phases in agile models tend to blur together more than in plan-driven models, and there tends to be less documentation of work products required, the basic idea being that code is what is being produced and so documentation efforts should focus there. See the Agile Manifesto web page at http://agilemanifesto.org to get a good feel for the agile development model and goals.

We’ll take a look at several life cycle models, both plan-driven and agile, and compare them. There is no one best process for developing software. Each project must decide on the model that works best for its particular application and base that decision on the project domain, the size of the project, the experience of the team, and the timeline of the project.

### A Model That’s not a Model At All: Code and Fix

The first model of software development we’ll talk about isn’t really a model at all. But it is what most of us do when we’re working on small projects by ourselves, or maybe with a single partner. It’s the *code and fix Model*.

The code and fix model, shown in Figure 2-1, is often used in lieu of actual project management. In this model there are no formal requirements, no required documentation, no quality assurance or formal testing, and release is haphazard at best. Don’t even think about effort estimates or schedules when using this model.

Code and fix says take a minimal amount of time to understand the problem and then start coding. Compile your code and try it out. If it doesn’t work, fix the first problem you see and try it again. Continue this cycle of type-compile-run-fix until the program does what you want with no fatal errors and then ship it.

Every programmer knows this model. We’ve all used it way more than once, and it actually works in certain circumstances: for quick, disposable tasks. For example, it works well for proof-of-concept programs. There’s no maintenance involved and the model works well for small, single-person programs. It is, however, a *very dangerous* model for any other kind of program.

With no real mention of configuration management, little in the way of testing, no architectural planning, and probably little more than a desk check of the program for a code review this model is good for quick and dirty prototypes and really nothing more. Software created using this model will be small, short on user interface niceties, and idiosyncratic.

---

¹Paulk, M. C. *The Capability Maturity Model: Guidelines for Improving the Software Process*. (Reading, MA: Addison-Wesley, 1995.)