KEVLIN HENNEY

Software development can be more of an education than you might first think. Kevlin Henney studies what there is to learn

Learning curve

A BUG IS NO MORE AND NO LESS THAN A SOFTWARE defect. However, the less harsh and less direct name "bug" masks the nature of the beast and helps to ease the conscience of programmers and the organisations around them. The term also helps plays down the frustrations of software users, to the point that defects have become accepted as a normal and reasonable state of affairs. The word has left its jargon origins and joined mainstream English.

OK, so now we know what a bug is, what is it worth? In practice, a bug has negative value: it uses up good will; it costs money and time to discover it and uncover its modus operandi; it takes time and concentration away from whatever else we were doing.

However, in principle, a bug also offers us a learning experience. There is the potential for something positive to come from it: we learn about a particular usage or situation we had previously overlooked; we learn more about a requirement that was poorly articulated or tacitly assumed; we learn that a piece of code we assumed to be correct was subtly problematic or obviously wrong – deeply and systemically or because of a simple thinko or typo – and we learn how to fix it. We also learn that there might be an opportunity for us to change our habits and choice of practices to reduce the likelihood of such defects occurring again in future.

With the exception of the last learning point, all of these are the direct and immediate lessons we take from bug reports. They are fairly local in their effect and often we look no further. But the last point is the meta-lesson that could really add the most value. A development model that emphasises debugging after the fact over practices that reduce enbugging in the first place has its priorities back to front and represents a failure of learning.

Learning to develop software

The idea of learning having a central position in software development is more than a metaphor: it identifies one dominant aspect of the software development process. Pundits are often happy to brand software developers as knowledge workers. One image this inspires is of knowledge as some kind of artefact or workpiece with the software developer as the artisan standing over it, crafting it with all manner of knowledge lathes and conceptual hammers. However, knowledge is not physical enough to support such a metaphor. It cannot simply be shipped and shaped as iron or wood.

The acquisition and presentation of knowledge is all part of the established topic of learning. Learning involves accumulation, consolidation, exploration, articulation and feedback, all of which can be seen in effective software development processes. The emphasis on an iterative and incremental approach is something that distinguishes agile development processes from more bureaucratic and master-planned processes. A cyclic and cumulative approach reflects a learning process.

When "learning" is normally mentioned in the context of software development, it is assumed to refer to development skills. This is the gap filled and fulfilled by books, training courses and so on. But more generally learning and the expression of the knowledge acquired defines the axis along which software development runs: what we learn is embodied and revealed in the code behind the software [1].

For example, there is learning about the domain in which a piece of software runs or is to run, and the specific needs that define the scope and purpose of the software and any changes to it. This responsibility is not only restricted to someone with the title "analyst"; it applies to all those who are involved in formulating the software, including the party for whom the software is intended. A common criticism of software customers is that they do not know what they want even though they know they want it; it is easier to reason about this perception from the perspective of learning. Unless there is a process that encourages learning on all sides, how else is the knowledge going to emerge clearly? Miracles and master plans? Feedback obviously plays a significant role in all this, and shortening long feedback loops is one of the optimisations that can make any approach more effective.

Design education

Software development is a design-based activity. Some people assume that design means "drawings and documents about the software, but

not the software". However, this is not a particularly useful or accurate reflection of what design-based professions do. Design is about the creation and expression of structure, physical or virtual, which fulfils a number of goals and satisfies a number of constraints. It is a creative and intentional act with many modes of expression and levels of detail.

Goals and constraints can be decomposed recursively from the highest-level of granularity into finer levels of detail. Requirements exist at the application level with respect to its users, but requirements also exist locally in the way that one method uses another [2]. However, there is no simple sausage machine that successfully turns a handful of use cases captured in natural language into an effective running system. Design is involved in everything from framing the requirements to demonstrating the effectiveness of a running system.

Design embraces all the kinds of expression of structure from whiteboard sketches in ad hoc UML to lines of code in Java. This means that the view of code as no more than an implementation detail – and therefore one that is not a proper concern or activity of design – is one of those unhelpful myths that has dogged software development for too long, and yet fails to stand up to close scrutiny.

Code is a formal notation used to express structure and execution in a virtual world – very little code, even that written in something as metal hugging as assembler, can claim to be describing structures and execution in a strictly physical world. Code creates and inhabits a designed domain [3]. A significant number of decisions that affect the final software are made at the keyboard in a code editor. This is not necessarily always a good thing, but it is necessarily inevitable. Any sustainably realistic view of development needs to do more than just acknowledge this inevitability; it needs to take advantage of it.

To understand the inevitability, consider detail: all design activities are in some way based on abstraction, but not all are or aim to be complete. A package diagram sketched out on a whiteboard may be accurate, but it does not offer a complete view of what has been or is to be developed – if it did, it would be unreadable and useless, defeating the very purpose of using such a sketch. By contrast, the definition of a wire-level protocol needs to be both accurate and precise for it to be of use – a sentence such as "...and the bits ought to be laid out in some order or other" is unlikely to be of much use. Code, in its many forms, is still abstract, but it demands a high level of both precision and accuracy. There is no cheating on the completeness of the detail needed – the variation across various programming languages and infrastructures is with respect to the amount of detail needed, not its completeness.

Of course, this is not to say that all designers are – or should be – equally good at all kinds of design. Such a statement would not be well supported by the facts. However, it does indicate that design is a very broad church that cuts across many different concepts and practices; successful design involves playing on and bridging these differences to best effect.

Shaker loops

Successful design is inevitably incremental, although some steps might be larger than others. It inevitably involves acceptance of change, and therefore renewal and removal, rather than the obstruction of change. Of course, just to be clear, acceptance of change is not the same as an absence of stability. And, to take us back to where we started, successful design does not rely on software defects as its primary source of feedback.

Design is both a feedforward and a feedback process, which makes it a dialogue: between people; between people and the design; between people and the results of the design. This is where learning fits in. So there is a need to shorten long feedback loops, but there is also a need to increase signal to noise ratio by reducing interference – continuous unregulated feedback is noisy rather than helpful.

Active testing

To take an example people often associate with the issue of bugs, consider the role of testing in software development. Is testing, as is often articulated, concerned only with the discovery of defects in software as built? Is testing, as is sometimes assumed, the sole preserve of a testing department and individuals with "tester" in their job title? Is testing, as expressed in many traditional development processes, something that necessarily falls in the closing phases of the lifecycle after all the "development" has been done?

If testing is divorced from other activities in software development and placed towards the end of the development lifecycle, carried out only by individuals who were not involved in the design of the system, its main contribution to the software is pretty much limited to one thing: defect reports. This is certainly better than nothing (and also better than many companies manage) but it represents a wasted opportunity for learning and a poor distribution of responsibilities, schedule and money. In such lifecycles, testing is consigned to play a passive rather than active role, a role that cannot influence the framing of requirements, the design decisions, the coding guidelines or the team and its individuals until after the fact. If there is any genuine learning, it will typically be localised and lost or, at best, deferred to the next project.

As I said, a wasted opportunity. It is generally considered wiser to close the stable door before the horses bolt. Testing offers a form of empirical feedback that can be used to drive design, clarifying requirements and providing a health check for code. Running testing along the main axis of software development and across more than one development role shakes up the pipeline model of traditional development processes. It closes many of the open feedback loops and shortens many of the others, while at the same time ensuring that the signal is stronger than the noise. Instead of treating testing as a critical passenger, shoved into the backseat, it becomes an informative driver. Testing ceases merely to be a synonym for bug hunting and becomes a proactive form of learning. ■

References

1. Kevlin Henney, "Code versus Software", artima.com, August 2004, www.artima.com/weblogs/viewpost.jsp?thread=67178
2. Kevlin Henney, "Inside Requirements", *Application Development Advisor*, May 2003.
3. Michael Jackson, *Problem Frames*, Addison-Wesley, 2001.

Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com