Books
Articles
Interviews
Presentations
About Me
Contact Me

IEEE SOFTWARE

Best Practices

IEEE Software, Vol. 14, No. 2, March/April 1997

Software's Ten Essentials

Virtually every backpacker, rock climber, and recreational hiker in the Pacific Northwest is familiar with the Seattle Mountaineers' list of "Ten Essentials": extra clothing, extra food, sunglasses, knife, firestarter, first aid kit, waterproof matches, flashlight, map, and compass.

The Ten Essentials are the end-product of years of hard-won experience. They are intended to help mountaineers avoid getting into trouble in the first place, and, if that doesn't work, to minimize the damage. No experienced mountaineer would go into the mountains without the Ten Essentials.

Experienced software developers have also accumulated years of hard-won experience. Our software adventures often contain more uncharted paths and dangerous territory than a simple hike in the woods does, and so I propose a list of Ten Essentials for software projects.

Software's Ten Essentials

A *Product Specification* is a software project's compass. Without one, you can perform the work of Hercules and still not produce a working product because the work in aggregate hasn't been aimed in any particular direction. Without good direction, any individual's work can go the wrong direction and different people can work at cross purposes.

With today's highly interactive systems it is becoming increasingly difficult to capture the essence of a product specification without constructing a *Detailed User Interface Prototype*. Static paper documentation often cannot adequately describe the intended look and feel of a product. If the product specification is the compass, the detailed user interface prototype is the trail map that points out the hills and valleys, groomed trails and portions of the software outing that will require special skills.

A beneficial side effect of user interface prototyping is that it can be an effective way of lighting a fire under both the customer and the development team. Visibly working software is good for customer and developer morale. A user interface prototype isn't working software, but it looks like working software, and it can have almost the same effect.

No experienced hiker would think of going on a long hike without sufficient food, water, and clothing. On a software project, a *Realistic Schedule* provides the essential planning foundation for adequate staffing, adequate quality assurance activities, and in general the appropriate level of formality in the project's software processes. Every fall we hear of hikers trapped in the woods by an unexpected snowstorm. Every spring we hear about a software product that was supposed to ship on January 1 but which doesn't actually ship until many months later. Basing a software project on an unrealistic schedule and the insufficient staffing and technical planning that result from it is tantamount to heading into the woods in November without a warm jacket.

If a hiker gets into trouble, it's useful to know that a person can go for days without food but not without water. A successful software project establishes *Explicit Priorities*, so that if it gets

into trouble it knows which features are essential and which can be jettisoned. Explicit priorities help to avoid the problem of wanting all possible features with the best quality in the shortest time with the least effort. Setting "I want it all" priorities is tantamount to setting no priorities at all. They provide no guidance when the project needs to make tough choices. Explicit priorities make the tough choices easier.

A common theme running through the Ten Essentials is that of hoping for the best but preparing for the worst. You wouldn't go hiking if you expected to break your leg, and you wouldn't start a software project if you expected it to run 300 percent over budget. In spite of your best hopes, however, you'd be foolish to go hiking without adequately preparing for the risks inherent in the activity. *Active Risk Management* is also a key component of successful software projects. As Tom Gilb says, if you do not actively attack the risks on your project, they will actively attack you.

A *Quality Assurance Plan* is the software project's first aid kit. The first priority in first aid is avoiding doing anything that will require you to use the first aid kit. But even the most careful hikers sometimes get hurt, and in such a case a first aid kit is essential. Many software projects perform the moral equivalent of leaving the first aid kit in the car. By the time problems become too obvious to ignore, much of the damage has been done. Defects have been inserted into the product and not corrected during requirements and design activities. All that can be done at that point is to correct the defects at great cost during construction and system testing. A good quality assurance plan will orient the project toward detecting defects early, close to the point of insertion and not allow defects to infect work later in the project.

For longer hikes, hikers have to file an itinerary. If the hikers file an itinerary for a 3 day hike and haven't signed out after 3 or 4 days, the Forest Service sends out a search party. Successful software projects use *Detailed Activity Lists.* These lists are typically comprised of tasks that last a few days each and that are considered to be either done or not done--not "90 percent done." Comparing the list of completed activities to the list of planned activities indicates whether a project is on time or needs to be rescued.

Software Configuration Management won't keep you warm and dry, but it will keep you from succumbing to some of the more dangerous software project risks. At the most basic level, software projects put source code under automated source code management. This prevents problems such as one developer inadvertently overwriting each other's work. Source code control is typically combined with an off-site backup plan so that if the server with the master sources crashes you're not left out in the cold.

At a more esoteric level, the most successful projects also put designs, requirements, and project planning materials under configuration management. When this is done, a change in the schedule or budget requires explicit approval and notification of the concerned parties. This helps to keep schedule and budget related decisions visible and prevents hundreds of small changes from quietly accumulating into large schedule and budget overruns.

Sometimes you'll see a hiker with a 20-year old backpack patched together with so much duct tape and twine that you can't make out the original backpack; that's what software systems developed without an explicit focus on *Software Architecture* look like. Internally, software architecture promotes consistent design and implementation approaches, which in turn facilitate future corrections and extensions. Externally, the most visible aspect of explicit software architecture is its support for consistent user interfaces. Consistency is a generally desirable characteristic that you attain almost automatically when you have good architecture and only with great difficulty when you don't.

One of the thorniest implementation problems is the problem of integrating software components that were not designed with integration in mind. An explicit *Integration Plan* is therefore the last of the Ten Essentials. With a good integration plan such as the Daily Build process (see this column in IEEE Software, July 1996), you can almost forget that integration tends to be a troublesome issue. Without an integration plan, you can enter an extended integration, test, bug-fix cycle that exposes so many defects that it can kill the project.

Software's Ten Essentials
1. A product specification 2. A detailed user interface prototype 3. A realistic schedule

4. Explicit priorities
5. Active risk management
6. A quality assurance plan
7. Detailed activity lists
8. Software configuration management
9. Software architecture
10. An integration plan

Other Essentials

Several organizations have published similar lists of software project essentials. The Software Project Manager's Network publishes a "Project Breathalyzer," which is a ten question test designed to determine whether a project should be on the road. The test is available on the Internet from http://www.spmn.com. The Standish Group published a report titled "Charting the Seas of Information Technology" which included a list of the top 10 success factors for MIS projects. The key process areas required to advance from Level 1 to Level 2 of the Software Engineering Institute's Capability Maturity Model might also be considered "essentials." You can read about those in *Capability Maturity Model for Software, Version 1.1* by Mark C. Paulk, et al, which is downloadable from the SEI's website at http://www.sei.cmu.edu.

Editor: Steve McConnell, Construx Software, 11820 Northup Way #E200, Bellevue, WA 98005.
E-mail: steve.mcconnell@construx.com - WWW: http://www.construx.com/stevemcc/

Email me at stevemcc@construx.com.

Construx
Software Development Best Practices