

Long considered an afterthought, software maintenance is easiest and most effective when built into a system from the ground up.

BY PAUL STACHOUR AND DAVID COLLIER-BROWN

You Don't Know Jack about Software Maintenance

EVERYONE KNOWS MAINTENANCE is difficult and boring, and therefore avoids doing it. It doesn't help that many pointy-haired bosses (PHBs) say things like:

"No one needs to do maintenance—that's a waste of time."

"Get the software out now; we can decide what its real function is later."

"Do the hardware first, without thinking about the software."

"Don't allow any room or facility for expansion. You can decide later how to sandwich the changes in."

These statements are a fair description of development

during the last boom, and not too far from what many of us are doing today. This is not a good thing: when you hit the first bug, all the time you may have "saved" by ignoring the need to do maintenance will be gone.

During a previous boom, General Electric designed a mainframe that it claimed would be sufficient for all the computer uses in Boston, and would never need to be shut down for repair or for software tweaks. The machine it eventually built wasn't nearly big enough, but it did succeed at running continuously without need for hardware or software changes.

Today we have a distributed network of computers provided by thousands of businesses, sufficient for everyone in at least North America, if not the world. Still, we must keep shutting down individual parts of the network to repair or change the software. We do so because we've forgotten how to do software maintenance.

What is Software Maintenance?

Software maintenance is not like hardware maintenance, which is the return of the item to its original state. Software maintenance involves moving an item away from its original state. It encompasses all activities associated with the process of changing software. That includes everything associated with "bug fixes," functional and performance enhancements, providing backward compatibility, updating its algorithm, covering up hardware errors, creating user-interface access methods, and other cosmetic changes.

In software, adding a six-lane automobile expressway to a railroad bridge is considered maintenance—and it would be particularly valuable if you could do it without stopping the train traffic.

Is it possible to design software so it can be maintained in this way? Yes, it is. So, why don't we?

The Four Horsemen of the Apocalypse

There are four approaches to software



maintenance: traditional, never, discrete, and continuous—or, perhaps, war, famine, plague, and death. In any case, 3.5 of them are terrible ideas.

Traditional (or “everyone’s first project”). This one is easy: don’t even think about the possibility of maintenance. Hard-code constants, avoid subroutines, use all global variables, use short and non-meaningful variable names. In other words, make it difficult to change any one thing without changing everything. Everyone knows examples of this approach—and the PHBs who thoughtlessly push you into it, usually because of schedule pressures.

Trying to maintain this kind of software is like fighting a war. The enemy fights back! It particularly fights back when you have to change interfaces, and you find you’ve only changed some of the copies.

Never. The second approach is to decide upfront that maintenance will never occur. You simply write wonderful programs right from the start. This is actually credible in some embedded systems, which will be burned to ROM and never changed. Toasters, video games, and cruise missiles come to mind.

All you have to do is design perfect specifications and interfaces, and never change them. Change only the implementation, and then only for bug fixes before the product is released. The code quality is wildly better than it is for the traditional approach, but never quite good enough to avoid change completely.

Even for very simple embedded sys-

tems, the specification and designs aren’t quite good enough, so in practice the specification is frozen while it’s still faulty. This is often because it cannot be validated, so you can’t tell if it’s faulty until too late. Then the specification is not adhered to when code is written, so you can’t prove the program follows the specification, much less prove it’s correct. So, you test until the program is late, and then ship. Some months later you replace it as a complete entity, by sending out new ROMs. This is the typical history of video games, washing machines, and embedded systems from the U.S. Department of Defense.

Discrete. The discrete change approach is the current state of practice: define hard-and-fast, highly configuration-controlled interfaces to elements of software, and regularly carry out massive all-at-once changes. Next, ship an entire new copy of the program, or a “patch” that silently replaces entire executables and libraries. (As we write this, a new copy of Open Office is asking us please to download it.)

In theory, the process accepts (reluctantly) the fact of change, keeps a parts list and tools list on every item, allows only preauthorized changes under strict configuration control, and forces all servers’/users’ changes to take place in one discrete step. In practice, the program is running multiple places, and each must kick off its users, do the upgrade, and then let them back on again. Change happens more often and in more places than predicted, all the components of an

item are not recorded, and patching is alive (and, unfortunately, thriving) because of the time lag for authorization and the rebuild time for the system.

Furthermore, while official interfaces are controlled, unofficial interfaces proliferate; and with C and older languages, data structures are so available that even when change is desired, too many functions “know” that the structure has a particular layout. When you change the data structure, some program or library that you didn’t even know existed starts to crash or return `ENOTSUP`. A mismatch between an older Linux kernel and newer `glibc` once had `getuid` returning “Operation not supported,” much to the surprise of the recipients.

Experience shows that it is completely unrealistic to expect all users to whom an interface is visible will be able to change at the same time. The result is that single-step changes cannot happen: multiple change interrelationships conflict, networks mean multiple versions are simultaneously current, and owners/users want to control change dates.

Vendors try to force discrete changes, but the changes actually spread through a population of computers in a wave over time. This is often likened to a plague, and is every bit as popular.

Customers use a variant of the “never” approach to software maintenance against the vendors of these plagues: they build a known working configuration, then “freeze and forget.” When an update is required, they build a completely new system from the ground up and freeze it. This works unless you get an urgent security patch, at which time you either ignore it or start a large unscheduled rebuild project.

Continuous change. At first, this approach to maintenance sounds like just running new code willy-nilly and watching what happens. We know at least one company that does just that: a newly logged-on user will unknowingly be running different code from everyone else. If it doesn’t work, the user’s system will either crash or be kicked off by the `sysadmin`, then will have to log back on and repeat the work using the previous version.

Real-world structure for managing interface changes.

```
struct item_loc_t {
    struct {
        unsigned short major; /* = 1 */
        unsigned short minor; /* = 0 */
    } version;
    unsigned part_no;
    unsigned quantity;
    struct location_t {
        char          state[4];
        char          city[8];
        unsigned warehouse;
        short         area;
        short         pigeonhole;
    } location;
    ...
}
```

However, that is not the real meaning of continuous. The real continuous approach comes from Multics, the machine that was never supposed to shut down and that used controlled, transparent change. The developers understood the only constant is change and that migration for hardware, software, and function during system operation is necessary. Therefore, the ability to change was designed from the very beginning.


Software in particular must be written to evolve as changes happen, using a weakly typed high-level language and, in older programs, a good macro assembler. No direct references are allowed to anything if they can be avoided. Every data structure is designed for expansion and self-identifying as to version. Every code segment is made self-identifying by the compiler or other construction procedure. Code and data are changeable on a per-command/process/system basis, and as few as possible copies of anything are kept, so single copies could be dynamically updated as necessary.

The most important thing is to manage interface changes. Even in the Multics days, it was easy to forget to change every single instance of an interface. Today, with distributed programs, changing all possible copies of an interface at once is going to be insanely difficult, if not flat-out impossible.


Who Does it Right?

BBN Technologies was the first company to perform continuous controlled change when they built the ARPANET backbone in 1969. They placed a 1-bit version number in every packet. If it changed from 0 to 1, it meant that the IMP (router) was to switch to a new version of its software and set the bit to 1 on every outgoing packet. This allowed the entire ARPANET to switch easily to new versions of the software without interrupting its operation. That was very important to the pre-TCP Internet, as it was quite experimental and suffered a considerable amount of change.

With Multics, the developers did all of these good things, the most important of which was the discipline used with data structures: if an interface took more than one parameter,



Software maintenance is not like hardware maintenance, which is the return of the item to its original state. Software maintenance involves moving an item away from its original state.



all the parameters were versioned by placing them in a structure with a version number. The caller set the version, and the recipient checked it. If it was completely obsolete, it was flatly rejected. If it was not quite current, it was processed differently, by being upgraded on input and probably downgraded on return.

This meant that many different versions of a program or kernel module could exist simultaneously, while upgrades took place at the user's convenience. It also meant that upgrades could happen automatically and that multiple sites, multiple suppliers, and networks didn't cause problems.

An example of a structure used by a U.S.-based warehousing company (translated to C from Multics PL/1) is illustrated in the accompanying box. The company bought a Canadian competitor and needed to add inter-country transfers, initially from three of its warehouses in border cities. This, in turn, required the state field to split into two parts:

```
char  country_code[4]
char  state_province[4];
```

To identify this, the company incremented the version number from 1.0 to 2.0 and arranged for the server to support both types. New clients used version 2.0 structures and were able to ship to Canada. Old ones continued to use version 1.0 structures. When the server received a type 1 structure, it used an "updater" subroutine that copied the data into a type 2 structure and set the country code to U.S.

In a more modern language, you would add a new subclass with a constructor that supports a country code, and update your new clients to use it. The process is this:

1. Update the server.
2. Change the clients that run in the three border-state warehouses. Now they can move items from U.S. to Canadian warehouses.
3. Deploy updated clients to those Canadian locations needing to move stock.
4. Update all of the U.S.-based clients at their leisure.

Using this approach, there is never a need to stop the whole system, only the individual copies, and that can be

scheduled around a business's convenience. The change can be immediate, or can wait for a suitable time.

Once the client updates have occurred, we simultaneously add a check to produce a server error message for anyone who accidentally uses an outdated U.S.-only version of the client. This check is a bit like the "can't happen" case in an `else-if`: it's done to identify impossibly out-of-date calls. It fails conspicuously, and the system administrators can then hunt down and replace the ancient version of the program. This also discourages the unwise from permanently deferring fixes to their programs, much like the coarse version numbers on entire programs in present practice.

Modern Examples

This kind of fine-grain versioning is sometimes seen in more recent programs. Linkers are an example, as they read files containing numbered records, each of which identifies a particular kind of code or data. For example, a record number 7 might contain the information needed to link a subroutine call, containing items such as the name of the function to call and a space for an address. If the linker uses record types 1 through 34, and later needs to extend 7 for a new compiler, then create a type 35, use it for the new compiler, and schedule changes from type 7 to type 35 in all the other compilers, typically by announcing the date on which type 7 records would no longer be accepted.

Another example is in networking protocols such as IBM SMB (Server Message Block), used for Windows networking. It has both protocol versions and packet types that can be used exactly the same way as the record types of a linker.

Object languages can also support controlled maintenance by creating new versions as subclasses of the same parent. This is a slightly odd use of a subclass, as the variations you create aren't necessarily meant to persist, but you can go back and clean out unneeded variants later, after they're no longer in use.

With AJAX, a reasonably small client can be downloaded every time the program is run, thus allowing change without versioning. A larger client

would need only a simple versioning scheme, enough to allow it to be downloaded whenever it was out of date.

An elegant modern form of continuous maintenance exists in relational databases: one can always add columns to a relation, and there is a well-known value called null that stands for "no data." If the programs that use the database understand that any calculation with a null yields a null, then a new column can be added, programs changed to use it over some period of time, and the old column(s) filled with nulls. Once all the users of the old column are no more, as indicated by the column being null for some time, then the old column can be dropped.

Another elegant mechanism is a markup language such as SGML or XML, which can add or subtract attributes of a type at will. If you're careful to change the attribute name when the type changes, and if your XML processor understands that adding 3 to a null value is still null, you've an easy way to transfer and store mutating data.

Maintenance Isn't Hard, It's Easy


During the last boom, (author) Collier-Brown's team needed to create a single front end to multiple back ends, under the usual insane time pressures. The front end passed a few parameters and a C structure to the back ends, and the structure repeatedly needed to be changed for one or another of the back ends as they were developed.

Even when all the programs were on the same machine, the team couldn't change them simultaneously because they would have been forced to stop everything they were doing and apply a structure change. Therefore, the team started using version numbers. If a back end needed version 2.6 of the structure, it told the front end, which handed it the new one. If it could use only version 2.5, that's what it asked for. The team never had a "flag day" when all work stopped to apply an interface change. They could make those changes when they could schedule them.

Of course, the team did have to make the changes eventually, and

their management had to manage that, but they were able to make the changes when it wouldn't destroy our schedule. In an early precursor to test-directed design, they had a regression test that checked whether all the version numbers were up to date and warned them if updates were needed.

The first time the team avoided a flag day, they gained the few hours expended preparing for change. By the 12th time, they were winning big.

Maintenance really is easy. More importantly, investing time to prepare for it can save you and your management time in the most frantic of projects. 

Related articles on queue.acm.org

The Meaning of Maintenance

Kode Vicious

<http://queue.acm.org/detail.cfm?id=1594861>

The Long Road to 64 Bits

John Mashey

<http://queue.acm.org/detail.cfm?id=1165766>

A Conversation with David Brown

<http://queue.acm.org/detail.cfm?id=1165764>

Paul Stachour is a software engineer equally at home in development, quality assurance, and process. One of his focal areas is how to create correct, reliable, functional software in effective and efficient ways in many programming languages. Most of his work has been with life-, safety-, and security-critical applications from his home base in the Twin Cities of Minnesota.

David Collier-Brown is an author and systems programmer, formerly with Sun Microsystems, who mostly does performance and capacity work from his home in Toronto.