MS-DOS 6 Development Post-Mortem
Ben Slivka, Mike Dryfoos
May 3, 1993

Executive Summary

Lessons Learned

Details

Code reviews helped code quality/stability, but more design reviews were needed

One thing we focused on early was doing code reviews for all check ins, no matter how large or small. While this took more up-front time, it was a way to communicate coding techniques and guidelines, and improved the initial quality of all code in the project, and helped us meet our very aggressive dates. Code reviews also ensure that at least two people (the author and the reviewer) are familiar with **all** the code in the system, which ensures consistency of design and implementation. Indeed, this may be the **key** benefit of code reviews! We should continue this very valuable practice in future projects.

We need to do a better job, though, of design reviews before the coding starts. Some aspects of the product suffered from poorly optimized or excessively ad hoc design, and so became

sources of bugs, requiring time-consuming rework. We need to make our designs more explicit and give them greater review at the outset. The combination of code and design review will help ensure that our code is tight in both conception and mechanics.

Focus on robustness improved the product, but needed to be applied more thoroughly

When compressing a FAT drive (i.e., turning it into a DoubleSpace compressed drive), and when we shrink or grow a DoubleSpace Drive, we hook autoexec.bat so that in the case of a reboot we can regain control and continue the operation. We also make sure to disk I/O in such a way that we **can** continue without losing any data. This was very valuable in our product development, as it allowed us to break in to a Compress in Place (CIP) and see what was happening, possibly fixing problems and continuing without data loss. This also proved to be a very valuable (and highly demoable) customer feature!

Likewise the recovery capabilities of Memmaker have been very useful and important.

Unfortunately, it turns out we needed to be more thorough in applying the same standards to other areas of the product. We didn't do a good enough job supplying tools for data recovery and repair, nor did we do enough to permit users to access their compressed volumes when something has gone wrong internally, nor did we give adequate information to help users or support techs to solve problems when they arose. Some of the tools we did supply, such as Defrag and Dblspace /Chkdsk, have proven to have holes we should have closed up.

Future projects should continue to focus on safety features, as they speed development and are great features that have real end-user benefit. We need to make sure developers keep robustness and recoverability concerns in the forefront when designing and coding.

Used third-party resources well, but needed to review their contributions more carefully

We greatly enhanced our product through the creative leverage of third-party resources. In this manner, we were able to deliver a package offering great value with only a small investment in in-house development.

Unfortunately, the integration and quality control of the third-party additions were troublesome, largely because of that very lack of development attention. Without internally resources to look over the code and identify problems, we were at the mercy of our development partners for help. The response from our partners ranged from the very good at Helix to the very poor at Central Point. In the future, we need to ensure that we have some development as well as testing resources available to work directly with our partners, if we are to make sure their code upholds the standards we set for our own.

In addition, we need to be more thorough and concrete about specifying acceptance criteria in our acquisition contracts. Third-party code must be subject to review for overall quality, buildability, and localizability. Specific requirements must be

stated in these areas. Final acceptance must be withheld until these requirements are met. To do otherwise, as we now know from both MS-DOS 5 and 6, is to invite trouble and extra work.

Focus on the end date helped unify the team, but also led us to cut some corners

The product team really kept a strong focus on the ship date throughout the project. We all shared a clear understanding of where we were going, and kept in mind the tradeoffs we had to make to get there. The shared goal was a strong unifying force for the team.

Unfortunately, to a degree we became captives to the ship date. More time for creative testing and tracking down non-readily-reproducible bugs might have helped avoid some of the problems customers are now experiencing. While it isn't clear that we would have found or corrected any of the problems encountered since release, a less rushed atmosphere at the end would have given us more of a chance.

Beta team did a good job supporting both users and developers, but was not large enough

The beta team did an effective job of front-line support and service to the beta testers. Bug reports were screened effectively, keeping the developers from being inundated by duplicate and already-solved problems. Rollout of beta releases was quick and responsive to rapidly changing circumstances, as beta candidates were modified and bugs fixed.

But, due to the number of beta test sites, and the complexity of MS-DOS 6, the beta team was quickly overwhelmed with problem reports, and there were sometimes lags of one week or more before the development team got some reports, and when reports were bounced back to the beta team to acquire more information, there were sometimes delays of over a month. In future projects, we should ensure that the beta team is staffed adequately with very experienced PSS technicians -- those that we did have had a major impact on the productivity of the beta team.

Strong spirit of cooperation reigned throughout the project team

The different functional groups (development, program management, testing, documentation, etc.) within the project team worked together very smoothly. Lots of informal communication and cooperation, in the good Microsoft style, helped keep everyone productive, and the project running well. Different teams and team members worked jointly to set priorities and carry out tasks, without turf battles or excessive conflict.

Should have specified CVF format and DS APIs completely up front

We licensed code from Vertisoft, and then hopped into modifying and rewriting that code (both in the DoubleSpace Manager and the driver) **before** we had fully understood the compressed volume file layout. This approach led to several off-by-1 errors and disagreements between the device driver and the maintenance code, which were took a lot of time to find, debug, and fix.

We also did not add DBLSPACE.BIN API to do direct I/O to the compressed volume file, as asked for by Norton. Had we done so, we could have introduced the concept of mounting drives without creating a host drive in a future release, without causing MS-DOS 6-compatible disk utilities to break.

In the future, all media and interface standards should be spec'd up front, even if they are going to change over time, to improve the chance that different pieces of code will agree. We should also review these with key third-party ISVs and **listen** to their feedback.

Should have taught and enforced coding conventions and assertion checking

We had an early go at code reviews that focused on coding conventions, coding style, and assertion checking, but these were not enforced through the full cycle of the product. Hence we had code that was not maintainable (mysterious names, multi-page functions, duplicated code, duplicate constant definitions, lack of constant definitions, lack of asserts) that made it difficult and time consuming to make global changes when we were under extreme time pressure. There were several changes we made at the end that were complicated by the definition of the same concept in **multiple** locations, thus requiring us to search for and change much more code than otherwise would have been necessary.

We did not put assertion checking into the driver until our 2nd external beta, which delayed catching many bugs, including our off by 1 errors in our understanding of the MDFAT. We also did not check the areas of the CVF that the manager was writing to on grow/shrink operations, and so had to track down some very subtle bugs based on some very obscure symptoms. The DEBUG build actually had a bit too many debug "squirty outs" -- debugging trails written to a log file. We should have been more focused on producing a smaller, more meaningful set of information.

In the future, these standards should be written down and agreed to up front, and rigorously enforced. While Microsoft can often be a democracy, in matters of coding style, an exception should be made, and there should be one person who has final say in selecting and enforcing the coding conventions (most likely the development lead). This enforcement **must** occur through the life of the project, stopping only, perhaps, in the last month before you ship. "Hacking" in changes without regard to maintainability may **seem** expedient in the short term, but it will likely hurt you in your current product, and will definitely hurt you for future products.

As for assertion checking, you can never have too much, except as far as emitting too many debug trails. Especially in core code, you should have asserts that compute valid situations in more than one way, as a way to "assert the assert" -- i.e., to make sure the assertion checks do not have the same bugs that may exist in the code.

Should have focused on object-oriented data hiding

The split between the user interface of DoubleSpace Manager and the "Compression Engine Library" (CEL) was conceived to allow

a Windows-based UI to be written without requiring changes to the CEL. CEL would understand all the details of the CVF format, restartability, etc. Unfortunately, this concept was not carried out, and so major, major aspects of Restartability, Robustness, and CVF limits and properties are spread throughout both Manager and CEL.

In future, we should make sure all team members have practice at object-oriented design (focusing on data hiding and minimal interfaces), to ensure that code will be modular, decoupled, and maintainable. A 2-4 week teaching period where project is designed and implemented would be a very, very good thing.

Should have given developers important 3rd-party hardware and software

Some of the areas we had significant or repeated problems, such as removable hard drives, non-Microsoft networks, and add-on shells, become problems because developers don't use these things themselves. We typically rely on the beta test to reveal problems, but by the time the issues are revealed and understood, it is too late to make the design changes that might be necessary to accommodate them smoothly. We need to make sure more of these common add-ons are in the hands of developers from the beginning, so support for them is designed in from the beginning.

Should have increased testing focus on boundary conditions and error and stress scenarios

Many of the errors we encountered relate to the margins of normal operation, such as physical hardware errors and disk-full conditions. These have proven to be holes in the product, through which too many users have fallen.

More creative focus on stress, error, and boundary conditions would have made our product more robust. Achieving effective coverage in these areas typically requires more time and imagination than the more straightforward system testing we usually do. The testing and development teams need to work together to design the appropriate test cases. Thorough exercise of boundary and error conditions should remain in the hands of the test team, but developers can help out by doing more such testing up front.

In future, we should make sure the development team routinely includes some boundary case testing of new code. Developers should be required to produce mini test plans, along with the results of that testing, before checking code in.

Should have explored patent status thoroughly and immediately

We got surprised by the Stac Electronics lawsuit. We should have seen it coming, and we should have spent much more lawyer time digging up other relevant patents and looking at prior art.

In future, and projects which create new technology or license existing technology that has even the tiniest whiff of patents around it should call in the lawyers and spend some time digging up and examining patents. I know this is expensive, but especially for mass-market software, the downside can be

tremendous -- we've spent untold hours in development, marketing, and testing on responding to requests from legal in response to the Stac suit.

Should have done better PSS training, and predicted problem areas better

More advance PSS training from those closest to the product, on the areas we expected to be problematic, would have helped the support technicians deal with the initial load. Doing a really good job on this would require that we anticipate all the problems our users are going to have, which we have had only mixed success doing, but a more thoroughgoing effort should have been made at the outset. PSS needs to make an early commitment to advance training for as many techs who can be accommodated, and the product team needs to make delivering quality training a priority.

Many of the product team members participated directly in product support after release. This task was tackled with seriousness and commitment, despite the frustrating nature of the work. The team's efforts made a significant impact on the overall support effort.

<<< the end >>>