Astronomical Data Analysis Software and Systems IX
ASP Conference Series, Vol. 216, 2000
N. Manset, C. Veillet, and D. Crabtree, eds.

Using a Data-Driven Model for Instrument Software Development

D. A. Clarke, S. L. Allen

UCO/Lick Observatory, University of California, Santa Cruz, CA 95064

Abstract. In a data-driven approach to the development of instrument control software, we attempt to abstract from the code as many repetitive data structures and operational parameters as possible, storing these data in a relational database instead of in hard coding. Documentation and source code can then be generated from a single authoritative source: the database engine. In order to pursue this approach, we made significant changes in our software development process. We report on the degree to which the data-driven model (in which the database engine is an essential component in code development and deployment) has succeeded.

1. The Problem

If you look at the three highlighted regions in Figure 1, you will see a chunk of a GUI, a chunk of some formal documentation, and a chunk of some low-level C code. These might typically be written by two different programmers, and maybe a technical writer. You can also see that the same data appear in all three. The actual menu, page of text, and C code are merely thin, specific syntactic wrappers around an essential common core of information.

Using traditional software development methods, the material in these three forms is literally typed three times. This is the kind of mind-numbingly repetitive and boring work most prone to stupid typographical and scribal errors – some of which can have very subtle bad effects at run time, and be really hard to find.

Also, the three syntactic forms of this same information must be kept up-to-date and in synch with one another; and this effort is equally boring, tedious, endless, and very difficult to get right. The usual result is that drift sets in between the documentation and the code, and to a lesser extent between the lower levels of the code and surface applications such as GUIs. A lot of time is wasted finding out that programmer A changed a parameter and forgot to notify programmer B; and it's hard to maintain reliable code.

These problems, and the perception of a commonality of data, led us in 1995 to a preoccupation with the idea of abstracting these core data into a single authoritative online source – like a relational database. Applications could draw upon this database to *generate* documentation and even source code. With only one authoritative source of the information, programmers would not be chasing their tails trying to keep these different forms of the same data in synch; they could do more productive things with their time.

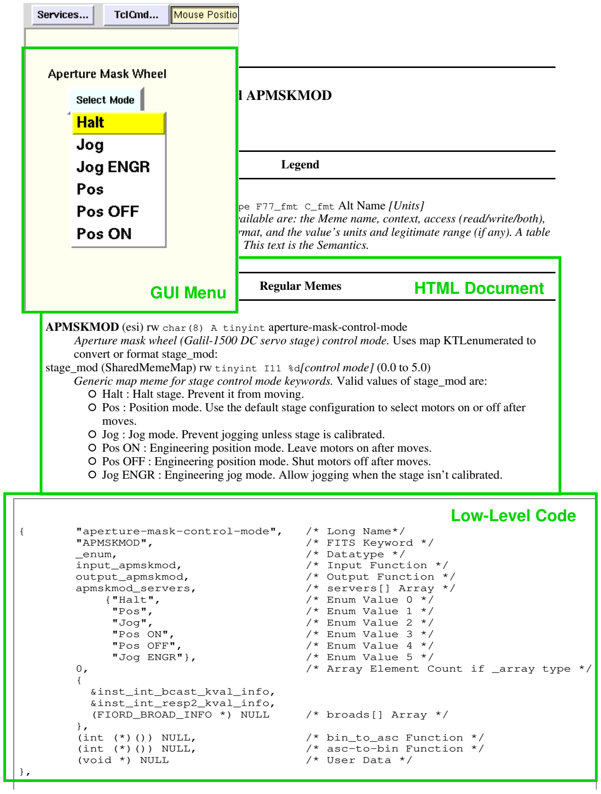

Figure 1. Repetition of data in software products.

In our ADASS96 paper (Clarke & Allen 1997) we argued that significant benefits might be realized by modeling FITS keywords as relational entities and using the resulting data plus a lightweight scripting language (Tcl/Tk) to generate products such as documentation and source code. This approach, we claimed, could eliminate much repetitive hand-coding, reducing both errors and development time. We were sufficiently convinced of these benefits that in the intervening years our software group adopted a data-driven development strategy.

Originally we applied this idea only to FITS keywords and their appearance in image headers, documentation, code, and instrument control GUIs. But over time, we ended up modeling far more than just FITS keywords, and we had to invent a more generic name than "keyword" for what was in our database. We came up with "meme", an in-house jargon term for a "unit of meaning". Thus, this database is now known as the Memes database.

2. A Partial Solution

We now routinely generate a certain percentage of our production instrument control code from the Memes database. Source code is generated and then correctly placed in the source tree by "make" targets, and then checked into

CVS. It can be checked out and built without regeneration. A variety of files are generated: C source code, Galil controller source code, and configuration files of various kinds.

How much code are we able to generate? For a typical instrument, about 60 files, about 14.5 thousand lines, or 1 MB of source. This is source that is not handcrafted, repeatedly tweaked, or repeatedly proofread by humans. We also generate some 20k lines of documentation and a whole slew of diagrams (mostly in the form of mapped GIFs for use on web pages) which document the structure of the control system. Again for a typical large instrument, about 2 MB of documentary files.

Aside from code and documentation generation, we've found that a whole set of lightweight applications has spun off from the database project. These utilities and tools are light and easy to develop because of their lack of internal, nitty, repetitive instrument-specific data structures; these structures have been externalized into the database. Our poster paper (Allen & Clarke 2000) in these Proceedings describes a few of these tools.

3. Costs and Benefits

Obviously, no change of methodology is without cost. There was a significant investment in database schema design, data entry, and the core applications which generate code and documentation. This effort, however, served three different instruments and therefore, being split-funded, was easier to justify than it would have been for any one instrument. We were under pressure to improve our productivity, so there was strong motivation to consider a change of methods.

Instrument programmers had to be trained to enter data using forms and to use a more rigorous make/install scheme than had previously existed. Integrating the generated files into the make scheme for the source tree also posed some challenges. We were developing the code generation system at the same time as the actual instruments and their code, so there were moments of confusion.

We removed nitty detail from many areas, but in compensation we concentrated all that complexity (and some fragility) into the generator apps. They became a vulnerable point; if they break, multiple people can be inconvenienced.

However, by taking these risks and accepting these costs, we did reduce the amount of time spent writing, proofreading, and cross-checking a fairly large volume of obscure low-level code and configuration files. The last minute discovery of discrepancies between one person's work and another's diminished noticeably, and communication between programmers was facilitated by their sharing a common "notebook" (i.e. the database) of all the parameters and structures that define the control system.

Documentation which, under normal circumstances, would never have been written (or would have been written once in the form of specifications and then never updated to match as-built reality) was easily generated, and re-generated, as the instrument software evolved. We were able to ship more complete documentation with these projects than had previously been possible.

We mentioned earlier the surprising proliferation of very lightweight applications which used the same Memes database as the code and documentation generators. These applications were light and cheap to write because they were free of repetitive, instrument-specific data structures and data; also, by their nature they were immediately applicable to any instrument developed using the Memes database. Re-use of code suddenly involved not even the most trivial re-write, but just a change of command-line arguments.

The benefit which was perhaps least expected but most important in the long run, was that the process of modeling our instruments, keywords, control system parameters, controller design, information flow, etc. in a formal way using relational concepts was itself very revealing. Inconsistencies, design flaws, and misconceptions suddenly leapt out at us and we caught, and fixed, many long-standing problems in our legacy code.

4. Conclusion

In summary, this was a rather radical and dangerous experiment. We invested a lot of time and effort in a fairly complicated retooling, a revision of the way we think about coding. It has taken three years to determine whether this was a mistake or a success. In that time we have shipped two instruments with software developed by this new method. From a software standpoint, the commissioning of the ESI spectrograph in late August went extraordinarily well; no observing time was lost due to software problems. It's unlikely that we could have produced software of this quality without our new tools and methods.

At this point we can say cautiously that the data-driven approach has been a success. Our benefits have outweighed our costs; the period of heavy investment in the Memes project is over, but the benefits are ongoing and can be shared by every instrument we build from now on.

Acknowledgments. We'd like to thank the rest of the software group at Lick Observatory, who were willing and long-suffering guinea pigs for this experiment; and especially our boss, who was willing to bet the farm on our bright idea.

References

Allen, S. L., & Clarke, D. A. 2000, this volume, 339

Clarke, D. A., & Allen, S. L. 1997, in ASP Conf. Ser., Vol. 125, Astronomical Data Analysis Software and Systems VI, ed. G. Hunt & H. E. Payne (San Francisco: ASP), "Practical Applications of a Relational Database of FITS Keywords"

We invite the reader to visit our ADASS99 Web Pages[1], and to find out more about the Tcl/Tk scripting language[2] and how we use it at Lick Observatory.[3]

[1] `http://www.ucolick.org/software/adass1999`

[2] `http://www.tcltk.com/`

[3] `http://www.ucolick.org/software/Tcl`