CMU-CS-84-127

ZOG and the USS CARL VINSON: Lessons in System Development

Robert M. Akscyn and Donald L. McCracken

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

March 1984

Abstract. This paper contains recommendations for developing computer systems for other organizations using emerging technologies. These recommendations are based on our experience developing a computer-assisted management system for the USS CARL VINSON, a nuclear-powered aircraft carrier, during the past four years. We recommend that such projects be conducted in a highly cooperative manner between the users' organization and the developers' organization over a planned period not longer than 18 months, obtaining feedback via a series of instrumented prototypes that are exercised by both users and developers for actual tasks.

This work was supported by the Office of Naval Research under contract N00014-76-0874. It was also partially supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, the Defense Advanced Research Projects Agency, or the U.S. Government.

This paper is to appear in the Proceedings of the first IFIP Conference on Human-Computer Interaction (Interact '84).

Introduction

Since 1975, the ZOG Project at Carnegie-Mellon University has been studying problems of human-computer communication. This research has proceeded empirically via the development of a general-purpose interface system called ZOG--a system that combines features normally associated with operating systems, databases, and editors. ZOG provides rapid response (less than one second) to user selections for browsing in a large, network-structured database in which the nodes are formatted, display-oriented chunks. These nodes, called "frames", have a structure general enough to support a variety of applications such as those listed in the example ZOG frame shown in figure INTER121F below. In addition, any item in a frame can be used to activate a process. A more detailed description of ZOG can be found in [1] and [3].

```
Application areas supported by ZOG                                  Inter121


  1.-Documentation development and management

  2.-Training

  3.-Issue Analysis

  4.-Software development and management

  5.-Electronic communication (mail, bulletin boards, teleconfercing)

  6.-NOTE This frame is not yet identical with the original paper



                                                                  ↑. parent

                                                                  *. top

edit help back next prev top goto acc mark ret zog disp user find info win xchg
```

Figure 1: An example ZOG frame

In February of 1980, we were visited by Captain Richard Martin, who had recently been assigned to shepherd the USS CARL VINSON through its final construction and serve as the ship's commander for its first operational cruise. He was investigating ZOG's potential for information management on board the VINSON, a task he foresaw as being monumental. At the time, we were looking for a real-world application to drive the further development of ZOG, and we agreed to work with the VINSON to develop a ZOG-based management system. Between then and February 1983, we jointly developed a distributed version of ZOG that runs on a network of 28 powerful personal computers (PERQs).This system supports a number of management applications, including an expert system

that assists decision-making for the launch and recovery of the ship's aircraft. A description of the ZOG/ VINSON project is contained in [2].

On a higher plane, this project was an experiment in what might be called "direct technology transfer", in that it bypassed the traditional procurement mechanisms for placing such systems in operational military environments. For us the experience was a rich one, and we believe that some of the lessons we learned may be applicable to developing computer systems in other contexts. In this paper we have tried to encapsulate a number of these lessons in the form of recommendations. We believe they are most relevant to researchers-turned-project-managers who are trying to exploit emerging technologies and develop systems for operational use. Table 1 gives a preview of the recommendations.

R1: Let the requirements evolve over time
R2: Develop a broad, multi-level model of the system
R3: Make progress on multiple levels in parallel
R4: Devise effective strategies for users to employ prototypes
R5: Treat the users as part of the development team
R6: Build a system the users' organization can support
R7: Insist that the developers use the system being developed
R8: Collect performance data by instrumenting the system
R9: Don't start on a shoestring
R10: Don't try to do too much (or too little)
R11: Use scripts of user-level behavior for automatic testing
R12: Keep a running log of the most recent user actions
R13: Give documentors a central role in the project

Table 1: Recommendations for managing development

Recommendations based on experience

This section contains our recommendations for managing the computer system development process, based on our four years of experience with the ZOG/VINSON project. In the following section we will cover some recommendations based on hindsight, i.e., ones that we did not follow this time but would be inclined to follow in future projects.

R1: Let the requirements evolve over time

Our view of how system requirements should be specified contrasts sharply with conventional system development procedures. Traditionally, complete functional specifications for a system are written in the initial phase of its development. To our way of thinking, this strategy is not applicable to projects that are based on rapidly emerging technologies (e.g., distributed software). In such projects, requirements should not be too detailed or rigid at the beginning since it is impossible to accurately determine them at that time. This is so for several reasons.

First of all, the various technological components may not mature soon enough to permit them to be exploited. Researchers are somewhat prone to setting their sights too high in development projects--offering to deliver results which, in some cases, require solving genuine research problems. Some of the development may depend on systems (e.g., an operating system) being developed by other groups whose goals could diverge from yours at any time. Consequently, it is prudent to seek explicit commitment for support from these other groups. Even so, since such commitments do not reduce the technological risks involved, backup solutions should be planned for all the subsystems that are based on emerging technology (including the subsystems under your control). Our project contained an excellent illustration of this point. We had planned from the beginning to use an advanced network operating system being developed by another group at Carnegie-Mellon, but this group was not responsive to the urgent deadlines contained in our project. As a result, we were finally forced to switch to a less capable, but working, operating system at the eleventh hour -- an extremely disruptive move.

A second reason to develop the specifications over time is that users will discover new functions (and abandon old ones) as their appreciation of the technology's potential matures. This understanding takes time for the developers as well, since initially they are not well informed about the nature of work in the users' organization.

Finally, we believe that adopting such a development paradigm has important ramifications for the future of the system. Rather than viewing the system as an entity which gets "completed", the developers and the users begin to view the system in more evolutionary terms--i.e., that a crucial feature is its ability to adapt to the needs of the users even after the developers are no longer involved.

Obviously we are speaking of a tradeoff here, not a categorical imperative, but the basic point still remains: a specification document can become a security blanket to which the developers cling, rather than cope with the rich opportunities that progressively reveal themselves.

R2: Develop a broad, multi-level model of the system

We believe there is a substantial payoff in developing a broad conceptual model of the system. By providing the individual developers with a view of the total system, it helps them make design decisions with global tradeoffs in mind. It also provides them with a framework for discussing their results with others. In addition, it provides a structure for documenting not just the design of the system, but the tradeoffs made as it is developed.

We believe that this model should have a *multi-level* structure, for a number of reasons. First, computer systems in general have this structure. Consequently, a strong correspondence with the technological levels of computer systems promotes incorporating future advances in those technologies. The model we developed for our own work had seven levels: hardware (PERQs), operating system (POS), language (Pascal), software (ZOG), applications (management), content (ZOG frames), and maintenance (including training and continued development). This framework is discussed in [2].

A second reason for a multi-level structure is that users learn more readily if shielded from some of the system's complexity. A system in which each level makes minimal assumptions about other levels reduces the need for users (and developers) to know the details about those levels.

However, there is a serious danger that developers of lower levels will become too isolated from the needs of higher levels--a problem compounded by the fact that experience with the higher levels lags behind experience with the lower levels. The crux of the problem is that the design of the lower levels implicitly constrains the capabilities of the higher levels, yet it is the higher levels which are the "value producing" parts of the system. We feel there must be a conscious effort on the part of the developers to make the lower levels of the system serve the higher levels--otherwise the system the users see will not truly exploit the technology.

R3: Make progress on multiple levels in parallel

In our view, there is only one good way for the needs of the higher levels of the system to take shape so that they can guide the development of the lower levels: progress must occur on the various levels in parallel. To accomplish this, the development of the system must proceed iteratively by implementing a series of functioning prototypes, so that the developers of higher levels have a platform to work upon.

However, parallel development creates a number of decision problems with which the managers of

the development process must cope. One such problem is deciding when to stop development of the current version for a given level and move on to the next. These transitions, which generally occur from the bottom up, can involve fairly radical changes such as moving to a new machine or language -- making the subsequent transfer of higher levels relatively problematic. In our case, we created five major versions of the system (see the list below), necessitating four major transitions for the developers. All versions except the last one were "throwaways". We believe this shows that we were willing to pay very high costs to gain the benefits of parallel progress.

Jan 81	Decided to create a temporary environment to develop application software on the old VAX version of ZOG, rather than waiting for the PERQ version.
Nov 81	Decided to build an interim standalone version of ZOG on the PERQ, using the existing operating system (POS).
Apr 82	Decided to build an interim ZOG on an early version of the new operating system, in order to make progress with the ZOG network server software.
Sep 82	Converted interim ZOG to a later (but still inadequate) version of the new operating system (with significant changes from the early version of Apr 82).
Nov 82	Made major decision to abandon new operating system and revert back to the POS operating system. (Required a major effort to reimplement many components of the system -- especially the network servers).

We were never able to plan well for these transitions, but rather had to make the decisions dynamically, often on quite short notice. The problems of transition are compounded when the next version of the lower levels is not yet ready for higher levels to move aboard (perhaps because the leap is too large!), but yet the current version has problems that are severely constraining progress at the higher levels. This leads to a "schizophrenic" style of development, in which developers of lower levels must also work on "obsolete" versions for the good of the project (much to their consternation).

R4: Devise effective strategies for users to employ prototypes

In order to receive meaningful feedback from actual users, we recommend providing the users' organization, as early as possible, a prototype that can be used to perform actual tasks.

There are some natural tendencies to postpone or avoid installing prototypes in the users' organization. Users will resist adopting a system that is partially formed and has lots of problems, while developers will want to tinker endlessly with the system before exposing it to the real world. However, gaining experience via a series of prototypes helps users develop a feel for the technology. In addition to generating many suggestions for design change, this understanding will enable the

users to see for themselves how their current practices could be modified to better utilize the technology (you won't be able to see it for them). Developers, on the other hand, typically underestimate the contribution that users can make; users are often more creative than the developers!

R5: Treat the users as part of the development team

We believe that the users should be treated as part of the development team--after all, they are the only ones who can really know what is needed. This does not imply abdicating technical leadership, but it does mean accepting the users as equal partners. In other words, the relationship should be a peer relationship, like a marriage should be, with neither side trying to dominate the other. Although placing the users in the design loop makes iterative design more difficult, the potential benefit is that both sides accept responsibility for the success of the system. In our case, much of the high-level design of the system was done in joint sessions with officers from the ship. We also assimilated into our development group several Navy officers who were stationed with us for extended periods of time. These officers became valuable members of our team -- the project could not have been completed without them.

R6: Build a system the users' organization can support

We believe that the users' organization should be capable of supporting the system at the end of the project. This is especially important when the system is not a product supported by a commercial organization. However, the users' organization may be unaccustomed to this point of view and therefore be reluctant to assume responsibility for continued maintenance. Part of the problem seems to be that many people view software as a form of perpetual motion--they expect that once it is set in motion (developed), it will run forever without any additional input of energy (maintenance). Others believe that maintenance is a relatively inexpensive activity and can be performed by people without much experience. Our view, on the other hand, is that proper maintenance is quite difficult and finding programmers with the right "service orientation" is a real organizational challenge.

The issue of ongoing maintenance is such an important one that we recommend it be resolved in the initial project negotiations. If the users' organization is unwilling to commit "upfront" the appropriate resources to establish its own support group (one which will ultimately take over full responsibility for the system), the development group may not wish to participate in the project. The developers, of course, must share the task of grooming such a support group, and must be willing to provide the support themselves until the new support group is fully ready to take over.

R7: Insist that the developers use the system being developed

In our project, we made every attempt to use ourselves the same system we were developing for the users' organization. (This would not be possible for all development projects, but for us it was quite feasible due to the generality of the tools we were creating). Making ourselves guinea pigs was often an inconvenience, but we persevered because we were getting double mileage out of our time: we were getting our work done (though not as efficiently as we might have liked) and exercising the system at the same time. This exercising was remarkably good for turning up problems with the system -- problems that we otherwise might not have become aware of until the real users had tripped across them. Problems also tended to get fixed more rapidly, since they were holding up our *own* progress. A valuable side effect of using ZOG for our project work was the development of additional capabilities; for example, facilities for documentation and software management.

R8: Collect performance data by instrumenting the system

In addition to the anecdotal feedback from users and developer-users, there are other, more systematic, methods for obtaining data on the performance of the system. These methods are a valuable supplement to traditional feedback collection mechanisms because their precision can pinpoint problems that would otherwise be missed.

Based on our previous experience with mainframe versions of ZOG, we embedded within the system a number of data collection mechanisms for measuring the behavior of both the user and the system. A description of some of the mechanisms we developed and the rationale behind them is contained in [4].

For example, in our operating system, a software bug that causes the program to crash (a "runtime error") triggers a listing of the stack of procedure calls at the point of the error. Like a radiologist's x-ray, this listing can be very informative about the problem. We designed the system so that these listings would be automatically included in the performance data collected by the system. In addition, we sensitized early users to the potential value of these listings and encouraged them to record what behavior they were engaged in at the time of the error, since this information provided the appropriate context in which to interpret the listing.

R9: Don't start on a shoestring

There are a number of important issues about resources that arise in projects based on emerging technology. You must clearly acknowledge that such projects require an enormous investment in

time. If the developers are not prepared to spend a significant amount of time learning about the users' organization, or if the users' organization is not prepared to make available the time for meaningful cooperation, we would recommend against the project. Consequently, you should expect that the project will last at least 18 months; not much can happen in less time.

However, there is a flip side to this problem: taking too long. When the project goes beyond 3 years, you're no longer managing a project, you're running a business. Organizational factors such as personnel turnover and burnout can begin to dominate the project.

R10: Don't try to do too much (or too little)

There is a fine line to walk between trying to do too much versus missing golden opportunities that arise along the way. However, the greatest danger lies with trying to do too much, since the result is that nothing gets done well. Since it is such a common mistake to set goals that are too ambitious, we forced ourselves at the beginning of the project to narrow our objectives to something we could carry off with a reasonably high probability. In retrospect, it is fortunate that we did this, because we were then able to take on two additional tasks that presented themselves as irresistible opportunities along the way. One of these additions (the expert system for aiding launch and recovery of aircraft) became more important in many people's minds than our original management applications. Even so, the situation was a mixed blessing, since it is painfully apparent to us that our effort was spread so thin that all of the applications suffered to some extent.

Recommendations based on hindsight

Of course, not all of our recommendations are based on things we did right. In this section, we include recommendations that we would implement in future projects because our experience proved them to be valuable by their omission. (We have selected only several of the more interesting ones; the brevity of the list does not imply that we made very few mistakes in managing the project.)

R11: Use scripts of user-level behavior for automatic testing

It is very time-consuming to test a new version of a system that has enormous richness. We believe that many of the problems detected by users after the release of a version could be detected much further upstream by exercising the system under the control of scripts that simulate a user's behavior. Part of the difficulty of validating complex man-machine systems arises from the fact that many problems are not just functions of single user behavior, but also interactions of multiple users that arise probabilistically.

R12: Keep a running log of the most recent user actions

When an unrecoverable error occurs, users do not reliably recall their actions which led up to the problem. Rather than counting on users to document their behavior and forward it with the appropriate error listing to the developers, we believe it would be much more effective to have this context information collected automatically and stored with the other information about the error.

R13: Give documentors a central role in the project

Since the documentors have to work with an image of the system from the users' perspective, they are able to see problems with the system to which the programmers are blind. We found the documentation of the system to be a useful measure of the quality of the system design: unnecessary complexities in the system became apparent through the attempts to document them for the users. We recommend that the documentor be respected as a valuable source of feedback for the design process.

Another important function of documentors should be to maintain central control over messages the system displays to users. Messages should be centralized in one location--preferably a data file rather than a program file for ease of modification. We allowed individual programmers to author messages, and were surprised in retrospect to see how inconsistent they were. When programmers control the messages, documentors often end up having to write a whole paragraph in the user's guide to explain a confusing message, when a comprehensible message would have done the job alone.

Summary

We believe that the development process for computer systems utilizing emerging technologies must preserve considerably more flexibility than suggested by more traditional development strategies. This need must be recognized by all participants in such projects. Goals are likely to change significantly over the life of the project as the understanding of the technology and the users' needs matures. Therefore, we recommend that such projects be conducted in a highly cooperative manner between the users' organization and the developers' organization over a planned period not longer than 18 months, obtaining feedback via a series of instrumented prototypes that are exercised by both users and developers for actual tasks.

Acknowledgements

We wish to acknowledge the contributions of many people over the years. Those who have been involved with ZOG at CMU: Allen Newell, George Robertson, Kamila Robertson, Elise Yoder, Sandy Esch, Patty Nazarek, Angela Gugliotta, Marilyn Mantei, Kamesh Ramakrishna, Roy Taylor, Mark Fox,

and Andy Palay. Those officers from the USS CARL VINSON who worked with us at CMU: Mark Frost, Paul Fischbeck, Hal Powell, Russ Shoop, and Rich Anderson. Captain Richard Martin, Captain Tom Mercer, Cdr Ted Kral and other officers and crew of the USS CARL VINSON. And finally, Marvin Denicoff from the Office of Naval Research, our original ZOG research sponsor.

This work was supported by the Office of Naval Research under contract N00014-76-0874. It was also partially supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, the Defense Advanced Research Projects Agency, or the U.S. Government.

References

[1] McCracken, D. and Akscyn, R., Experience with the ZOG human-computer interface system, to appear in the International Journal of Man-Machine Studies (July 1984).

[2] Newell, A., McCracken, D., Robertson, G. and Akscyn, R., ZOG and the USS CARL VINSON, Computer Science Research Review, Carnegie-Mellon University (1981) 95-118.

[3] Robertson, G., McCracken, D. and Newell, A., The ZOG approach to man-machine communication, International Journal of Man-Machine Studies, 14 (1981) 461-488.

[4] Yoder, E., McCracken, D. and Akscyn, R., Instrumenting a human-computer interface for development and evaluation, in the proceedings of Interact '84.