Why is software development so difficult to manage?

Technical Report No.137

R. Barrett and D. B. Christianson

May 1992

Why is software development so difficult to manage?

Ruth Barrett and Bruce Christianson

School of Information Sciences, Hatfield Polytechnic
Herts AL10 9AB, England, Europe

Software is extremely complex, invisible, and easy to change. Because it's so plastic, it's tempting to make lots of changes. Software is also an amazingly unforgiving medium with which to work. Changes to software frequently have devastating effects which were not intended, fail to have the effects which were intended, take far longer than seems reasonable to make, and if made in sufficient number, can cause the whole product to fall apart in your hands.

Yet change is inevitable in any project. Why is managing a project which involves software different to managing any other project? Why is software so hard to get right and so difficult to change? What can we as managers do to facilitate?

Of course there are plenty of snake-oil merchants who will claim that they have the answer to these questions, whether it be CASE tools, formal methods, fourth generation languages, relational databases or integrated programming environments. These are all very worthy innovations, and properly used they may do a lot of good. But they won't turn managing a software development project into a routine engineering exercise.

In real life, there are no such easy answers. Millions of man years have been invested in software development, and if there were any sure-fire solutions we would know them by now. No one who discovered such valuable information would be able to keep it a secret for long. The fact that so much effort has been invested without anybody finding a way of making managing a software development project look much like managing other engineering projects leads us to the firm belief that no such unifying approach exists. Software is somehow essentially different from other design media from a management point of view and we have to begin by accepting that, and trying to understand why.

The first thing is to understand why software is so fragile. The reason is that all code has a great deal of hidden context. By this we mean that it isn't obvious, just by looking at a piece of code, what assumptions it makes. It's hard to tell what it relies upon to make it work. A change somewhere else, in a far off part of the program, may have the effect of changing the context for this part of the program. In consequence a change that would have been all right yesterday may have a dreadful side effect if it's made today. That effect in turn may not become apparent for several weeks, until a change is made somewhere else again.

It's almost never possible to make just one change. Usually a whole heap of concommitant changes have to be made together. In the worst case, some of the knock-on changes may be incompatible with each other. In this case the proposed change simply isn't viable. Even if the changes can all be made together, the result may be to change the context of other pieces of code so as to make them more rigid and more prone to error.

Software interfaces are particularly difficult to isolate. It is not usually easy to see exactly what is being interfaced with what, and therefore what assumptions are required for the interface to behave in the intended way. This is because software interfaces usually have a great deal of history-dependent state buried within them. The representation of this state is usually implicit, rather than explicit. This makes software interfaces particularly resistent to change.

There are two main defences against software fragility. The first is to keep change to a minimum. Think of software as being something that is hard to change and respond to pressures for changes accordingly. Fix the software specifications early in the development process, and resist committing to changes once design is underway. Even if the changes look straightforward. Even (indeed especially) if the code hasn't been written yet.

Changes in the specification force the designer to re-draw their mental map of the complex internal context dependencies in the software which they are designing. It's hard to do that. It's hard to re-check that everything is still consistent. It's even harder to do this several times in succession. It's almost impossible if the effects of each change also have to be communicated to several other people whose invisible context may have been affected.

The use of CASE tools and programming environments which support formal methods can help the designer here by automating much of the detailed conceptual checking and so increasing the designer's mental reach. But there is a danger that management will simply use this as an excuse to increase the amount of change imposed. Management must accept responsibility for constraining the amount and timing of change if the quality of the finished artifact is to be secured. They need to be very clear about the *real* requirements of a project and so avoid window dressing.

The second defence against fragility is a good design. A good design has a high degree of internal coherence. This conceptual integrity makes it easier to spot where related changes are required. A good design makes it easier to understand the context of a piece of code, and hence to identify the concommitant changes. The required design integrity cannot be achieved by team effort, it is not possible if too many people are involved too early on.

A good design is only arrived at through a thorough understanding of the application domain and its potential for change. The components and relationships of the application domain should be mirrored in the software structure. Changes in the functionality or data in the application should result in corresponding changes in the software components. This structural view of a software system is only part of the picture, but it goes a long way to capturing and controlling the complexity of a system. Of equal importance is the control of the system and the resulting state of the system at any point in time - this is the context sensitive information that is so hard to capture and reason about. This state information is what makes it so hard to decouple components in the design and specify the precise nature or timing of an interaction. The problem is magnified when the designer must also build in concurrency and interprocess communication.

The solution to this problem is to not to hide the important decisions at the lowest level, but to ensure the control is modelled at a level abstract enough for the behaviour of the software system to be validated. The safety and liveness

properties of the system can then be ensured. One other fly in the ointment is the treatment of errors or illegal states. All errors in input, whether human or data transmission errors, should be retried at the point at which they occur, and so only valid data is passed to the components in the design. Run-time errors within other components should be propogated to a component which is able to recover the legal state. This policy will ensure clear interfaces between modules. A third important, often glossed over, aspect of design is the allocation of resources such as processor time, memory, human effort, and the estimation of these resources in differing situations. It is these calculations which may determine the viability of alternative designs.

Good design therefore takes time. It requires a detailed consideration of alternatives. But design is often an invisible process. It doesn't seem to have a deliverable. As managers we get nervous with activities which we can't monitor. How do we know whether things are going well or badly? At least once code is being cut we can see something being produced - and once we start testing we seem to be on safer ground at last.

We dispute this. If writing software is an engineering process then it's a very peculiar one. It has no production phase. Once a program is ready to ship, it's all just 0's and 1's. Coding is just an advanced stage of design, where the detailed decisions about representation and manipulation of data are being made. Being in a hurry to force detailed decisions before deciding fundamental issues is not going to improve the chances of things working.

With testing, we are on even shakier ground. The vast complexity of most useful pieces of software means that it is not possible (even in principle) to test every path. The implicit historical state information hidden in most interfaces means that it isn't generally even possible to test whether a software component meets its specification. What is really being tested is the integrity of the design. In fact, the hardest part of testing is to get an error into the open where the cause (ie context) can be determined. Furthermore, design changes imposed as a consequence of detecting non-trivial bugs create further problems for the designer.

Ultimately, the only defence against failure is to have a good design from the beginning.

This design should have a high level of conceptual integrity, which should not be destroyed as detailed design of the representation of the data and individual algorithms are added. The conceptual integrity is ensured by using the same philosophy when decomposing the system and by ensuring that all components at one level are at the same level of abstraction. The decomposition should be based on data rather than functions because the data is more stable over time. The conceptual integrity is also ensured by being able to adequately specify the behaviour of the system in all circumstances. A design with a high level of conceptual integrity can be shared more easily among the development team, communicated to those responsible for finding bugs, and changed with a higher degree of confidence.

The need to communicate software design gives us a clue to how to manage the software development process more effectively. Design documents are an important project deliverable. They are essential for unambiguous communication between members of the team, and for handover to maintenance (ie post initial development change).

Here lies a great difficulty, what form shall these documents take? How can we capture the decisions taken, and those rejected, without the production of the documents becoming an end in itself? The most useful documents for the designer are those which allow him to express and validate his design. The most useful to the project manager are those that represent some completion point. We must not force the generation of design documents solely for management purposes. They must evolve as part of the design process, otherwise they will not accurately represent the internals of the design.

There are very many aspects to this internal design, to try to capture all decisions in one type of document is confusing and counter-productive. The structure of the system can be represented diagrammatically, but the diagrams will need supporting text. These diagrams can show the relationship between the individual hardware and software components, but are not helpful in showing the functionality. The design of the control of the system is more difficult to document, many decisions are made using application and language specific knowledge and rely heavily on the experience of the designer. Pseudocode is inadequate except for very small systems or for detailed design. Attempts to address this problem include formal languages and synchronisation diagrams.

It is very difficult, if not impossible, to show the complete traceability of requirements in the design, and this contributes to the difficulty in quantifying the impact of making a change in the software structure. Good code documentation and CASE tools to capture design decisions increase the maintainability of the software. We agree with the notion of "separation of concerns" in deriving different aspects of design and yet a change in requirements rarely has an isolated effect. We come to the conclusion that software must be designed to accommodate change, but that this change must be carefully controlled.

As managers we must commit ourselves to an understanding of the nature of design documents and how they evolve. We must become concerned with the process by which they are generated. We must use their contents to become familiar with the internal structure of the artifact. The pressure to get the external behaviour correct at the expense of the internal design must be resisted.

If managers insist on regarding the software which their project produces as a black box, then the software will remain invisible to them (and hence remain beyond their management control) right through the crucial specification and design stages. They must be aware of the consequences of the decisions made in these stages on the fragile software structure. They will then be in a position to impose some discipline on the software operations.

Otherwise, by the time the software enters the coding stage, it may be too late to save the project.