

Software Processes: Live and Let Die

What characterizes the optimal process for a software project? The author presents a reengineering case study to demonstrate that the answer lies in a holistic approach—one that carefully considers interactions between the tasks, processes, and people involved with the project.

Allan Baktoft Jakobsen, *MindMate*

Imagine your e-mail inbox overloading with messages complaining about a piece of software you support. Imagine that you made that software. Put yourself in the chair of the manager, looking at hundreds of problem reports, knowing that each will cost, on average, \$7,000. Think of the consequences for your company's image. What will the customers say?

This was the situation in a company I worked for some years ago: people were whispering in the corners, discussing explanations and scapegoats—half-hearted attempts to put out the fire. We were rapidly losing money and self-respect, until one manager stepped forward and admitted: “This piece of software is fatal for us. We must completely rebuild it—whatever it takes.” The developers in the trenches (I was one of them) were overwhelmed by management's sudden resolution.

Thus begins my story about the reengineering of a catastrophic software product, a product that caused so much damage that change was the paradigm and “doing it the old way” became the exception.

A Mission to Save Our Software

In a state of emergency, it is no longer politically incorrect to suggest radical changes. Normally, you're up against a wall

of people arguing, “At least we know what we have. If we change, we don't know what we'll get.” In our case, we knew what we had wasn't working, and we had no choice but to clean up the mess so the software would never bug us again.

However, that software included about 50,000 lines of Basic-like code with about 200 input procedures, each with up to 25 parameters. Management encouraged us to look at the Cleanroom concept for software development that promised zero defects. The goal was to use Cleanroom techniques to rebuild the software within six months, and the rumor quickly spread that zero defects was the project's success criteria.

Management hired in-house consultants from the smaller software consultancy company with which our company cooperated. Furthermore, they flew in two Cleanroom specialists to catalyze the adoption of ideas through an intensive course. Then, they ap-

In our first reengineering attempt, the agreed-upon process collapsed—the project came to a full stop, and nobody knew exactly why.

pointed a project leader—someone with a reputation for fire fighting. Above him, the department boss managed the budget and the contract with the house consultants, which was not an easy job, because having a close relation with the consultants' company sometimes meant the jobs were initiated based on goodwill rather than on a legal contract.

Management then chose four developers—one from the group that developed the original software, one experienced tester, and two newly hired people (and I was one of these four people). The two new developers screamed for changes, because their first task in the company was to maintain the software. They had many new ideas but only a limited knowledge of the software. They were the type of people Ichak Adizes would call *entrepreneurs* (see Table 1).¹ The person from the original group seemed embarrassed and thus entered the project in a silent, defeated manner. He, along with the tester, preferred to work toward concrete goals—they were the type of people Adizes would call *producers*. You know the type—those who get frustrated if they haven't coded anything by the end of the day.

The Challenge

If developing software in general represents a challenge to human enterprise, then reengineering is an extra challenge. In this project, the people who had tried building the system the first time had, after all, failed terribly. We realized early on that this wouldn't be an easy project.

First, we focused on the technical problems: What in the software made it faulty, and what was the software's intended behavior? Later, our focus shifted to how to transform the old software into a new and better product.

There were two points of view from which we could consider the project. We

could consider the task and focus on the project's inputs and outputs—the inputs being the old software and the specifications and the outputs being the new, improved code—or we could consider the process and focus on how to transform the input to the proper output. The latter point of view has, of course, more general interest, because a process should be reusable for other input problems.

I could write an entire article on specifying and testing software and another one on Cleanroom reengineering processes. However, something disastrous but fascinating happened during the project: in our first reengineering attempt, the agreed-upon process collapsed—the project came to a full stop, and nobody knew exactly why. We were stuck with not only a catastrophic software product but also a catastrophic software process. To continue, we had to develop a new process.

Reengineering the reengineering process (metareengineering!) revealed hidden variables of the project and gave a deeper insight into the mechanisms of software development. I came to realize that we neglected to consider a third, rather important point of view. We never considered the human aspect—how the individuals interacted with the process and tasks.² From that perspective, the types of personality described earlier were of great significance, and they helped explain why, three weeks into the project, the initial process collapsed. We had to determine which process would best suit a group of developers dominated by enthusiastic producers and entrepreneurs who were not experts in the technical domain but who were eager to constructively solve the problem.

Two Reengineering Processes

Figure 1 illustrates the two processes we used. First of all, both used not only the original specifications as input but also the code for the original system, because during the development and test phases, the initial group added missing functionality and patches to the code without documenting them in the high-level specifications. This was the fundamental specification problem that our reengineering had to solve. Secondly, various Cleanroom techniques, especially the black/state/clear box

Table 1

Adizes types

Type	Description
Entrepreneur	A visionary, likes to develop new ideas
Administrator	Requires structure and order, prefers working on multiple tasks
Producer	Requires concrete goals and products, prefers working on a single task
Integrator	Cares about social interactions, works best in a harmonious environment

Black Box, State Box, and Clear Box Concepts

A basic idea in Clearroom is that software is an information system that transforms a given input or stimulus into a specific output or response, which the history of previous stimuli determines (see Figure A).¹ Describing corresponding stimuli (for example, *press the button*), responses (*the light turns on*), stimuli histories (*plugin followed by power on*) is the system's black-box specification. We typically represent this in a table but can also do so graphically.

In the state-box description, the significant stimuli histories are replaced by a certain data structure in the system that captures the events. This could be a state machine or a number of flags reflecting what happened previously. There can be a number of possible state boxes to a certain black box—each is the result of the design decisions made.

The clear box describes the dynamics applied to the data structure, or the logic. The clear box is thus an executable system not very far from the code. It's a matter of picking an appropriate programming language.

An extended interpretation lets the clear box consist of a system of interacting black

boxes. This enables an iterative approach in which you can use the black box, state box, and clear box suite at different abstraction levels.

Reference

1. H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, San Diego, Calif., 1986.

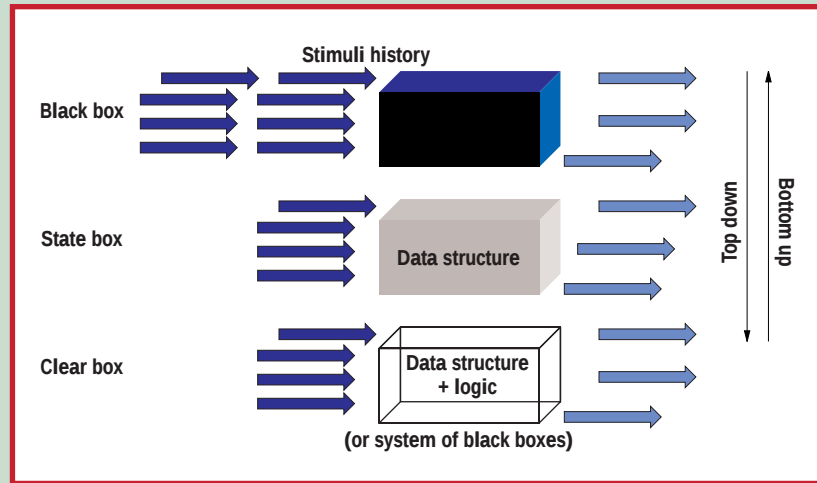


Figure A. The black box, state box, and clear box techniques.

concept, inspired both processes (see the related sidebar).

As Figure 1a shows, using Process A, you should derive one black box for the whole system (a total black box) before starting your redesign. The project then splits into two groups, a top-down group analyzing the original high-level specifications and a bottom-up group analyzing the code. Both should use the problem reports to derive the total black box—a table listing stimuli and responses with corresponding stimuli history. The true black box then emerges from the conflicts during the comparison.

After the analysis phase comes the synthesis (*analysis* means separating something in its components, and *synthesis* means putting something together). Based on the new total black box, you develop a new state box and finally a new clear box. You can organize the phases of specification, design, and coding as traditional waterfall development or alternatively using iteration.

In Process B, the initial analysis is much shorter (see Figure 1b). Instead of deriving the total black box, you should use the original high-level specifications together with the code's structure to propose an overall system

model or architecture. In other words, you create a clear box of many smaller black boxes.

Based on the hypothesis that this system model is reasonable, the work then involves excavating and rebuilding the boxes. If, for example, the model prescribes a box (or module) called an Operation Manager, you should consult the high-level specifications regarding the functionality. Then, analyze the code to locate the existing lines intended to fulfill the demands. Move around code lines to encapsulate the box.

This leaves you with three possible situations:

- If the clear box obviously fulfills the demands, directly make the corrections (changing the logic) and then reintegrate the modularized box into the software.
- If it's not obvious that the clear box fulfills the demands, but deriving the corresponding state box would make it obvious, then make the corrections to the state box and work out a new clear box. Then reintegrate the restructured box into the software.
- If it's necessary to derive both the state

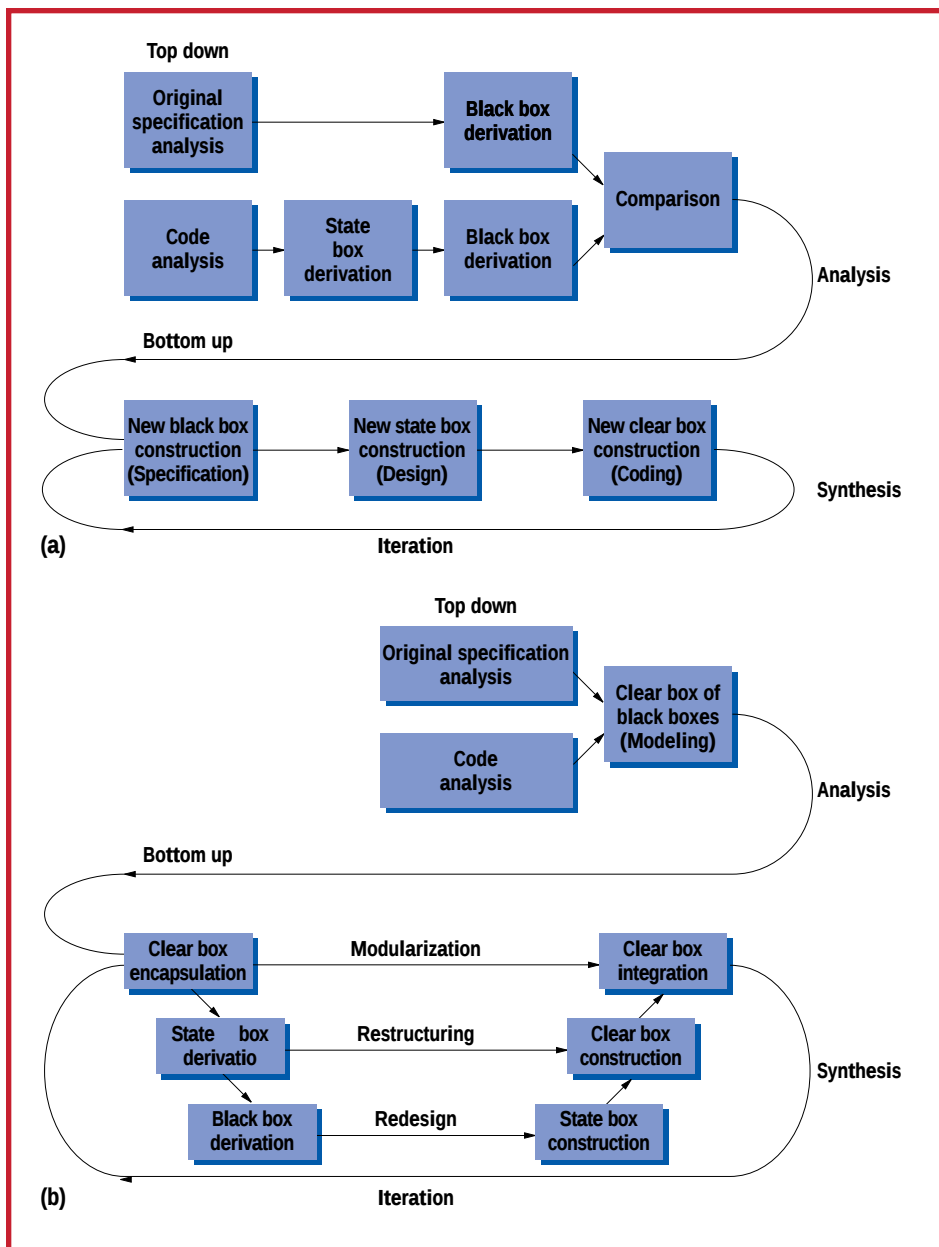


Figure 1. (a) Process A, based on deriving a black box before starting the design; (b) Process B, which involves developing a clear box of many black boxes.

box and the black box to check the functionality, then make the corrections at the black-box level and work out a new state box and clear box. Then reintegrate the redesigned box into the software.

The boxes should be taken out of the code and iteratively reintegrated starting with the most important boxes. Project participants should work together as one group in this process, using the original specifications and the problem reports to make the corrections.

Finding the Better Process

Our work began according to Process A, and we divided the group into two subgroups: a top-down and a bottom-up group. The project leader and the consultants de-

finied the process.

Soon after, a number of problems emerged: Deriving the black box for 50,000 lines of code is a huge and tedious job, and writing endless tables of stimuli and responses isn't very exciting. The best way to do it was continually a topic of discussion. The piles of papers grew and there was a big argument on when and how to compare the top-down and bottom-up black boxes. The comparisons caused conflicts between the people involved with the project.

The project leader arranged a crisis meeting. As developers, we were both angry and bored with Process A. The project leader asked about our progress during the first three weeks, and we estimated about 5%. Four people had completed 5% of the total work in three weeks, meaning the project would require 60 weeks rather than the allotted 24 weeks. We now had a fatal software system, a fatal process, and a tense atmosphere due to quarrels about the task and process.

A few days later, we came up with Process B. The consultants didn't like it much, but our department manager said it seemed more operational and gave us the go ahead to start the new process.

There was no trouble understanding our self-defined process, and we considered digging out the boxes from the old code and reconstructing them creative and even fun. Due to the iteration and reintegration, we produced reengineered software, which we tested nearly from the beginning. As we worked together, a more positive spirit emerged. With the small victories, success became evident and crystallized at the review meetings. The group became a team.

Six months later, the new software was put to the big system test. We did not meet the goal of zero defects, but the test identified only a handful of faults. The overall re-

sult was a success compared to normal software development in the company.

A Holistic Explanation: Why Software Is a Tough Business

This case study is complex because it contains three substories: the story of the task—that is, the reengineering of fatal software; the story about the two processes; and the story of the people involved in the project. Each story is interesting on its own: Reengineering is relevant when modernizing legacy systems, process improvements are becoming vital for the competitiveness of software companies, and motivating employees is crucial.

However, to understand why Process A failed while Process B succeeded requires considering the project from a combination of these three viewpoints. It requires a deep, holistic insight into software development.

The first explanation is that the process didn't match the task. The box techniques simply didn't scale up to 50,000 lines of code. The overall process would have taken too long, and the inputs did not suit the derivation process. Moreover, the early iteration of Process B made it more operational, because it produced better software almost from day one.

The second explanation is that the process didn't match the people involved. The developers were entrepreneurs or producers,¹ and Process A, with its tedious listing of stimuli and responses, required administrators. The biggest difference between the processes was probably the mixture of analysis and synthesis. With a group of producers without much technical knowledge, it seems important to get a process that will let them quickly get their hands on something concrete. The early synthesis made the difference here. Process B better matched the mixture of people, so teamwork had a better chance of success.

Finally, letting the developers define the process (as in Process B) ensured their understanding of it and provided motivation for creativity.

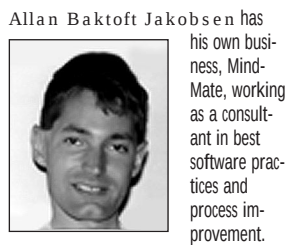
The people involved and their interest in the project was a key ingredient. In this case, the managers' main interest was to deliver a better product as soon as possible. How they delivered that product was of little importance. The consultants, on the

other hand, were more interested in the process. They could reuse the knowledge gained about reengineering processes in their own companies, but products and people are site-specific. Finally, the developers simply wanted an interesting job and an opportunity to succeed.

Understanding and optimizing these mechanisms is what software process tailoring is all about. You have to know your task and your people to tailor the process. Process A wasn't necessarily bad; it just didn't match the people involved with our project. The consultants later came back with an example showing that Process A worked fine for the reengineering of a cola vending machine. However, in our situation, the developers reacted using the only power they had—the control of the progress.

Focusing on processes alone (such as in the SW-CMM^{3,4}) does not provide enough information to understand and improve a software company's performance. Of course, that's easier said than done, because the various people in the project have various interests and priorities, which can bias the optimization. However, you still should always have a holistic view of software development to consider the developers involved in the project. Respecting their individuality and letting them define their own processes help ensure success; don't underestimate their power and abilities. 

About the Author



Allan Baktoft Jakobsen has his own business, MindMate, working as a consultant in best software practices and process improvement.

His work in the software industry has spanned maintenance, design, coding, test, and quality assurance. He received a BS in mathematics and physics and an MS in mathematics from Copenhagen University. Contact him at MindMate, Duevej 52, 2. tv, DK-2000 Frederiksberg, Denmark; baktoft@mindmate.dk.

References

1. I. Adizes, *How to Solve the Mismanagement Crisis*, MDOR Inst., Los Angeles, 1979.
2. J. Bach, "Enough about Processes: What We Need Are Heroes," *IEEE Software*, Vol. 12, No. 2, Mar./Apr. 1995, pp. 96–98.
3. W.S. Humphrey, *Managing Technical People*, Addison-Wesley, Reading, Mass., 1997.
4. J. Bach, "The Immaturity of the CMM," *American Programmer*, Sept. 1994, pp. 13–19.