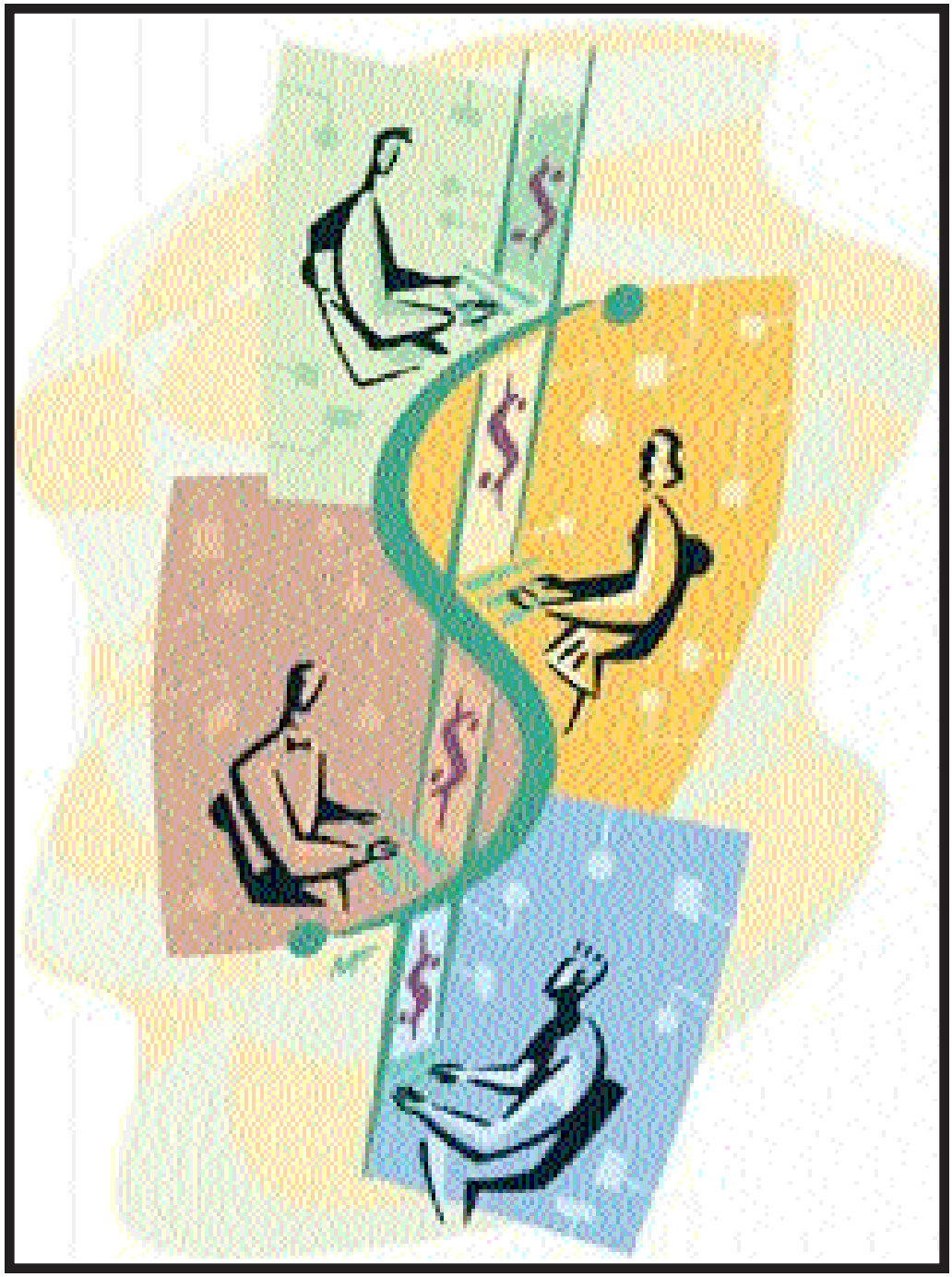
Chrysler Goes to "Extremes"



By the "C3 Team"

At Chrysler, Objects Pay

Chrysler Comprehensive Compensation (C3) was launched in May 1997. A little over a year before that, the project had been declared a failure and all code thrown away. Using Kent Beck's Extreme Programming methodology, Chrysler started over from scratch and delivered a very successful result. C3 pays some 10,000 monthly-paid employees and is being extended to support the biweekly-paid and weekly-paid populations. The team believes that its use of Extreme Programming made it all possible.

Extreme Programming

Extreme Programming rests on the values of *simplicity, communication, testing,* and *aggressiveness.* In this article, we'll comment briefly on simplicity, testing, and aggressiveness, while looking primarily at communication, the basis of our planning and tracking process.

Simplicity We can start with just one of Chrysler's pay populations, to keep things simple. But we have to be able to pay all the populations, with all their complexity. We're afraid that if we don't get all the requirements down, and if we don't build for the future, we may paint ourselves into a corner.

> Extreme programmers do the simplest thing that could possibly work.

We do not build generality into the system in expectation of future requirements. We build the simplest objects that can support the feature we're working on right now.

> Extreme programmers leave the system in the simplest condition possible.

Having built these simple objects, we refactor our code to eliminate any redundancy and ugliness from the code just installed.

These two rules together keep our objects well factored, containing only what we really need. When new requirements arise, the code is lean, mean, and easy to extend.

Yes, this goes against age-old programming lore: Get all your requirements up front; build general code to support the future. This thinking no longer applies in the flexible world of objects. You go fastest with the least code to drag around; with properly factored objects, the risk of cornering yourself is eliminated.

We have been developing C3 for over two years—in production for over one—and we have never once wished we had done something sooner or more generally. In fact we have many times wished we had built even less into the system, which would have let us go faster—additional function often gets in the way when the real requirement arrives.

We tried it, we know it's true. You go fastest when you "do the simplest thing that could possibly work."

Communication: Customer/Developer Developers are afraid that customers will demand everything at once; customers are afraid we won't get enough done; managers are afraid they won't know what's really going on. And we're all afraid of too much overhead.

"Do the simplest thing" also applies to communication. Extreme Programming helps us communicate better than most projects, while avoiding delay and overhead.

It all starts with the customer. In Extreme Programming, customers must be part of the project; they have the final say on what is needed and how good it has to be. Customers are part of the team throughout development.

The usual relationship between customer and developer can become adversarial: Customers demand all the features they might want, and developers resist making changes to make the deadline. C3 built healthy power sharing by considering just four measurable variables:

- ***scope***—what is done and what remains to be done
- ***quality***—correctness and other such measures
- ***resources***—personnel, equipment, space
- ***time***—duration of the project

If any three of these variables are chosen, we can figure out the fourth. Normally, customers define scope and quality, and resources are given; we figure out how long the project will take. Here's how.

Three Out of Four Ain't Bad Customers define *scope* with user stories, which are like use cases. They write each story on a separate card. Stories should encompass from one to three weeks of ideal development time; we learn to size them by experience.

Customers also define *quality.* They define functional tests that the system must accomplish before it can be released. By doing more testing of more important capabilities, customers have complete control over system quality.

With explicit public tests, everyone knows whether we are meeting the required quality level. We can't accidentally let quality decline because scope has increased or time has decreased.

Now we know scope, and quality, and resources. We need to predict what we're going to do and how long it will take, but we aren't sure the customers really know what they want and will ask for everything. We're afraid they'll change everything later, and that no one will understand the impact of change on the schedule.

Commitment scheduling. We do *commitment scheduling* both before development starts and regularly thereafter. We take all the story cards and spread them out on the table. We go through the cards, giving each an estimate of 1, 2, or 3 weeks of "ideal engineering time." Once we have estimated all cards, we arrange them according to priority, divide them into groups encompassing three weeks' work, count up the weeks, adjust for load factor, and come up with our delivery date.

But we don't know whether the customers even have all the stories yet. The developers' estimates might be wrong. We can't guarantee that everything will turn out according to the commitment scheduling prediction.

All these concerns are valid. We accept that our commitment schedule is an estimate, and at first it might not be a very good one. We commit to refining that estimate and publishing the result over the course of the project.

But even at the beginning, an estimate made by the actual developers is better than any date made up by someone else. Estimates get better as we go along, and we use the commitment schedule as part of our reporting process. It converges on reality and rapidly builds trust between customers and developers.

Iteration planning. To make sure developers know what to do, and that customers see how tasks relate back to stories, we work in three-week iterations and do *iteration planning* on the first Monday of each iteration. Customers select the user stories for the iteration and write them on the whiteboard. We discuss each story to make sure we understand it. Developers then write the engineering tasks for the story on the whiteboard. Once we have all the tasks, developers sign up for the tasks they want to do and estimate how much time they expect the task to take. If the estimates add up to less than the total engineering time available, then the customers add more stories; if the estimates add up to more, then the customers remove the lowest-priority stories. At the end of the iteration planning meeting, we all know what we have to do for that iteration, and everyone has a good overall picture of the next three weeks.

Each iteration corresponds to one of the three-week periods from the commitment schedule. As the iterations go by, we get immediate feedback on how the schedule is holding up. Because we redo the commitment schedule every three iterations, each subsequent schedule is based on more experience in estimating, and each is much more accurate than the one before.

> To go fast, we build only what we really need, thus keeping the system lean and mean.

Now we get down to work on the engineering tasks for the iteration. Communication remains central to what we do.

Communication with the Customer. We now need to assure that the developers know the details of what to do, and it doesn't take too long for them to find out, so that no one loses touch with the process.

Developers work in a single large room that we designed for ourselves. The room is set up for pair programming—two developers sit together to write all production code—and the customer space is right next to the developers. Communication between customers and developers couldn't be easier. When we have a question, we just walk over and ask. The key is immediacy: You can ask a question at any time, and get an answer right away.

We don't write memos back and forth, we sit down and talk. We have informal sessions, a couple of people resolving some issue, all the time. We have a daily stand-up meeting where developers and customers stand in a circle and give a brief progress report. This makes sure everyone always knows and agrees with what's going on, at a very low cost.

Communication with Management. We need to make sure we know how well we're doing, and that those higher up in the organization know how well we're doing. But we don't want a lot of fancy show and tell to waste time and obscure what's really going. So we do our reporting, all the way up to the vice-presidential level, in terms of the four variables. We tell everyone the same story:

- ***Scope.*** What percentage of the story cards has been completed, and what percentage of the way through the schedule are we?
- ***Quality.*** Show the graph of the functional test scores and whether they're improving and are converging on 100%.
- ***Resources.*** Do we have the personnel and other resources called for in our original estimates?
- ***Time.*** Give the current expected date for release from the most recent commitment schedule.

That's really all there is to reporting. The essence of our project status can be reported in four simple paragraphs, based on the four variables.

Communication Developer to Developer. We need to make sure that going fast will not mean that code will be put in willy-nilly, without enough planning or design—especially that there will not be code in the system that only one person understands, maybe even code that *no* one understands. Because we add function so quickly, we must ensure quality and understanding through some simple communication practices.

The Business Case

The C3 system will allow Payroll Services and IS to more easily manage the requirements for accurate and timely service to its 86,000 employees by reducing the duplication of effort the legacy systems require. Chrysler divides its employees into four groups for payroll purposes; each group is paid with a separate payroll system.

The hourly system pays 60,000 union-represented employees each week. The salary system pays 16,000 union and nonunion employees every other week. The executive system pays 10,000 management and technical employees once a month. The incentive compensation system pays 1,500 nonunion upper-management and international employees once a month. The corporate payroll department is responsible for the hourly, salary, and executive payrolls; the human resources group is responsible for the incentive compensation payroll.

The payroll department's three systems are twenty years old and showing it. Designed when a user interface meant an eighty-character punch card, each system requires a separate programming staff to maintain it and a separate customer staff to use it.

Quest for New System

In the early 1990s, the Payroll Services Department and the Information Services group decided to replace the three systems under their control with a unified system. The C3 team first tried a payroll package from a leading vendor. It couldn't handle the complexities of Chrysler's pay rules, and further analysis showed that no package could. The only option was to design and write a new payroll system.

Advantages of C3

- **Simplified movement between payroll systems.** A complex and often manual procedure is required to move employees between payroll systems. C3 eliminates this procedure, since there will only be one system.
- **Improved quality of manual input.** Manual input is currently written on forms and sent out for keypunch. C3 allows direct entry and immediate editing of data through GUIs.
- **Elimination of paper and microfiche reports.** Payment history can be viewed online.
- **Automation of manual procedures.** Many processes now done manually are automated in C3.
- **Better support for decision making.** C3 stores earnings and deduction information at the finest granularity. Reporting is done by adding up detail instead of backing down from aggregate values.
- **Simplified external interfaces.** Systems providing input to payroll now divide their data into separate feeds for each payroll system and in return receive separate reconciliation reports. Transactions sent to the wrong payroll system require manual correction in the payroll department and may result in the employee being incorrectly paid. Systems that feed payroll can send their files to a single point and C3 will find the employee's record regardless of pay frequency.
- **Opportunity to improve external interfaces.** The core of the legacy systems is its flat file masters and unit record transactions. However the systems that interact with payroll upgraded their technology, they couldn't alter their interfaces. While C3 supports the legacy master files and transactions, it does so only as a convenience. C3 can accept input from almost any source, including directly reading the other system's relational table or a Web-based GUI using CORBA.

By the end of October, the salary system's 16,000 employees will be paid by the C3 system. They will be joining the 10,000 executive system employees C3 has been paying since May of 1997. It is expected that the remaining 60,000 employees will move to C3 in mid-1999.

Products & Services Used

- Sun workstations running Solaris V2.5 with OpenWin/CDE
- PC workstations running Win95
- VisualWorks Smalltalk V2.5.1 with ENVY V3.01 on Sun server Enterprise 2 (development)
- GemStone V4.1.4.1 running on Sun server Enterprise 3000
- Kent Beck Testing Framework
- Refactoring Browser (UIUC)
- Block Profiler (Jim Haungs)
- BSI tax package (MicroFocus COBOL) statically linked into the VisualWorks virtual machine
- C, ProC, MicroFocus Cobol interface programs
- Sybase Net Gateway to MVS with CICS and DB2 (legacy input)
- Interfaced to KBMS expert system module (wage attachment)
- BeeperStrategy object beeps support in times of stress

To go fast, we have to know what we're going to do *before* we do it. Each chunk of development starts with a CRC-card design session. Several developers attend this session, and for anything interesting, customers attend as well. At the end of the CRC session, the customers know we understand what has to be done, and several developers know how the new features will fit into the system.

To go fast, the system must be kept lean and mean. When we put in our simple code, we then refactor to keep the whole system as simple as possible, making it easy to understand at all times.

We go fastest when we work in pairs. Every line of production code must be written by two people sitting together at the same terminal. This gives fast progress with high quality on everything we do. Plus, there are always at least two developers intimately familiar with any part of the system.

To go fast, we have to be able to change any objects that support the feature we're building. This means that any devel-

opment pair must read and edit code created by any other. So we all code exactly the same way: We name classes the same way, name methods the same way, even format our code exactly the same way. All the code looks familiar to everyone and is easy for everyone to understand.

Finally, there are extensive tests for the whole system, amounting to sample code showing how everything is supposed to work. When we need to learn (or be reminded) how something works, we can review the tests as well as the actual usage of the objects.

Testing

It is critical that, while going fast, we maintain quality. When we evolve the system to new capabilities, we don't want to break things that used to work.

We have over 3000 unit tests, testing every method that could possibly fail. We always write unit tests before releasing any code, usually before even writing the code. Our complete suite of unit tests, exercising the entire domain, runs in less than ten minutes, so we can afford to run all the tests every time anyone releases any code. And we do: We only release code when all the unit tests in the entire system run at 100%. We know we didn't break anything, which increases our confidence and lets us be aggressive in making necessary changes.

> We only release code when all the unit tests in the entire system run at 100%.

Customers are rightly concerned that we won't understand the requirements, or that we will make mistakes, or break things as we go along. Customers have responsibility for release, and they don't want to make a mistake.

Our customers own over six hundred functional tests, which are the final certification of the system. Developers provided the tools for building and running the tests, but customers define the tests and make the final decision for production release by examining the test results.

Each functional test has a corresponding user story. The test describes an employee, pays that employee, and checks tens to hundreds of results. Groups of tests are organized into suites, with each suite testing development from one project iteration.

We graph functional test scores every day, showing green for correct results and red for incorrect. Anyone can see from anywhere in the room how well we're moving toward function complete.

Aggressiveness

With Extreme Programming, simplicity plus communication plus testing yields aggressiveness.

Testing helps keep the system simple: We can always refactor, using the tests to make sure everything is working. The system stays simple, easy to understand and change.

Communication lets us know exactly what has to be done, and what everyone else is up to. Since we all work the same way, we can quickly edit any objects as we build what we need. There's no waiting for features needed in some other class, so we move quickly.

Simplicity is the core of aggressiveness: It's easy to be aggressive when you can understand what you're doing.

Conclusion

Currently, C3 is supporting Chrysler's monthly-paid employees and developing the next payment population, the bi-weekly. We maintain a single source: All new development is integrated weekly into the production system.

The C3 team found our old waterfall methodology too complex and cumbersome. Seeing the shortcomings of that approach and knowing it wouldn't get the job done, we were ready for Extreme Programming.

Extreme Programming is a great approach for teams implementing object-based applications. Object-orientation lends itself well to evolutionary development of systems. With CRC, new team members, developers, and customers quickly learn the meaning of C3's key objects. They are able to join the team and immediately contribute. Our team members' experience ranges from less than one year to over thirty years. None of us would consider using any other methodology.

We have focused on communication here, but many Extreme Programming practices have contributed to C3's success:

Old way	Extreme way
Limited Customer Contact	Dedicated Customers on Team
No metaphor	Good metaphor
Central up-front design	Open evolving design
Build for the future	Evolve to the future, just in time
Complex implementation	Radical simplicity
Developers in isolation	Pair programming
Tasks assigned	Tasks self-chosen
Infrequent big-bang integration	Continuous integration
GUI-driven	Model driven
Fear	Aggressiveness
Ad-hoc workspace testing	Unit / Functional Testing
Limited top-down communication	Communicate, communicate, communicate

Using the process described, the C3 team was able to start over on a very difficult problem and deliver a high-quality application on time and within budget. The combination of simplicity, communication, testing, and aggressiveness, applied by a disciplined team, gave the best results—and the most fun—any of us has ever seen.

Ann Anderson, Ralph Beattie, Kent Beck, David Bryant, Marie DeArment, Martin Fowler, Margaret Fronczak, Rich Garzaniti, Dennis Gore, Brian Hacker, Chet Hendrickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, and Don Thomas are the C3 Team, Chrysler Corporation.