Redesigning Air Traffic Control: An Exercise in Software Design

This case study demonstrates how basic software engineering techniques can make a complex system dramatically simpler. The authors describe lessons learned from reverse engineering an air traffic control system with a variety of tools and redesigning it to be smaller, simpler, and more flexible.

Daniel Jackson and John Chapin, *MIT Lab for Computer Science*

Many managers believe that improved process is the key to better software, and that technology is a second-order effect. We disagree. In this project, we dramatically simplified a complex software system, using basic software engineering techniques of the sort commonly taught in university courses but still not widely used in industry. The case study's original purpose was to explore the power of advanced techniques such as object modeling; we were surprised to discover that basic techniques provided a major improvement.

We took a component of a deployed air traffic control system written in about 80,000 lines of C++ code, replaced it with a new version in Java about one-fifth of the size, and demonstrated that the system still performed its primary functions. Although our new version only performs the original component's essential functions, we could extend it to cover the full functionality without substantially changing its architecture. In this article, we explain how we achieved this simplification and what lessons we drew from it, in particular for industrial practice and computer science education. (See the "Organizing and Reverse Engineering" sidebar for a discussion of how we organized the case study and reverse engineered the code.)

Overview of CTAS

The Center/TRACON Automation System (CTAS) is a suite of tools to help controllers manage air traffic flow at large airports. (Don't confuse CTAS with its anagram TCAS, which is a different system installed on-board aircraft that warns pilots of impending collisions.)

In the US, the rate at which aircraft can land at airports is the limiting factor in air traffic flow. CTAS increases the landing rate through automated planning. As input, CTAS receives the location, velocity, and flight plans of all aircraft near an airport, along with weather data, information about available runways and standard landing patterns, and controller commands. CTAS combines this information with models of the descent rates and other characteristics

0740-7459/00/$10.00 © 2000 IEEE

The case study ran as a one-semester graduate seminar entitled "6.894: Workshop in Software Design" (see http://sdg.lcs.mit.edu/~dnj/6894). Twelve students and three faculty members participated.

Organization

We started by reverse engineering the existing system. Based on the resulting list of problems and proposed solutions, we decided to reimplement the CM completely rather than modify the existing code. After substantial design work, the students spent three weeks in implementation and at the end successfully demonstrated basic system operation with the new CM.

Written in Java, the new component had about 50 classes and 10,000 lines of code. Handling of messages depended on a script about 1,800 lines long (of which 1,450 came verbatim from the previous C version of the CM). One of the teams built a compiler especially for the project that processed this script into about 2,000 lines of Java. The compiler itself was 10,000 lines of C++; its cost was not justified within the scope of the project, but might have paid off had we reimplemented the CM in its entirety. A separate paper describes the message scripting language and compiler.[1]

Reverse-Engineering Efforts

The case study began with an effort to understand the existing design of CTAS in general and the CM in particular. We used our ignorance of the existing design as an opportunity to experiment with several reverse-engineering tools. Of the lessons this experience provided, the most interesting perhaps was realizing how dramatically coding style affects the utility of analysis tools.

The documentation for CTAS includes motivation and architecture overview (http://ctas.arc.nasa.gov), software structures (www.ctas-techxfer.com), user manuals, and research papers on the underlying algorithms.[2] However, there appears to be no document that explains in high-level terms what the system computes or what assumptions it makes about its environment. Nor is there a design document that describes the relationship between the CTAS components: how they communicate, what services they offer, and so forth. We were forced to infer this information from the code, a challenge common to many commercial development efforts.

We used three off-the-shelf tools and built two others. We also spent considerable time simply reading the code, a task much eased by NASA's consistent naming and code layout standards.

Imagix

This commercial visualization tool constructs a cross-reference database from the source code and uses it to generate sophisticated reports and diagrams (www.imagix.com). We used it to generate a variety of call graphs. Its ability to abstract by showing calls between files rather than individual procedures was essential. It could handle the entire CM, and we generated some useful call graphs from it. We tried to use its impressive array of cross-referencing relations to compute an approximate object model from the header files, but were not successful.

Lackwit

This research tool uses type inference to analyze data structures in large C programs.[3] Lackwit generated a database representing the entire CM in about 80 Mbytes, and we used it successfully to answer queries about where in the code particular data structures were used. We also used it to construct a call graph showing only those procedures involved in the direct handling of an aircraft record.

Lackwit can make semantic distinctions that syntactic tools such as Imagix cannot. For example, one query asked where values held in a particular integer variable might flow. This variable held an identifier for a Route Analyzer process. The variables and data structure fields that Lackwit identified could not have been found with grep, because although many of these had a name ending "ra_index", not all did, and their type was not sufficient to distinguish them.

Unfortunately, a bug in Lackwit prevented us from analyzing static procedures (of which several were crucial),

of specific aircraft to accurately predict aircraft trajectories as much as 40 minutes in advance. This information feeds into dynamic-planning algorithms that suggest a landing sequence that minimizes unused landing slots. A CTAS installation at Dallas/Fort Worth Airport (DFW) has improved the sustained landing rate by 10%—a major success.

CTAS contains two primary tools. Low-altitude controllers who manage the airspace near an airport use the Final Approach Spacing Tool (FAST), while high-altitude controllers who manage aircraft further away use the Traffic Management Advisor (TMA). Both tools involve largely the same set of software components.

Figure 1 shows sample output from TMA, a timeline for aircraft crossing the boundary from the high-altitude domain to the low-altitude domain.

Figure 2 shows the architecture of CTAS in the TMA configuration. The Communications Manager (CM) sits at the center, where it acts as a message switch moving data among the other components, and maintains the database of aircraft information (position and velocity, aircraft type, and so forth). The Input Source Manager (ISM) collates input data streams such as radar feeds and flight plans. The algorithmic processes are

and Lackwit's handling of type casts was not as good as advertised.

CodeSurfer

This program slicer was in beta release (www.grammatech.com/ products/codesurfer/codesurfer.html) at the time of our project. CodeSurfer offered the most powerful features of the three off-the-shelf tools: it can generate program slices and display them graphically or textually. This would have let us show, for example, the code that might affect the value of a field in the aircraft database. Unfortunately, the CM proved too large for CodeSurfer to analyze directly.

Although CodeSurfer does not scale to the same size systems as Lackwit does, it provides more detailed semantic information than the type inference algorithms used in Lackwit. For example, it is flow sensitive. Furthermore, we would probably have had more success were it not for some flaws in the beta release that have now been corrected.

Concordance generator

One team of students constructed an ad hoc tool that generated a Web-viewable function concordance. When a function name was entered into the HTML form, the tool displayed an entry giving the function's arguments and results, a list of calling and called functions, and frequently a two-line specification. We cross-linked these entries to the code for easy navigation. The tool also generated diagrams showing call graphs of various forms; functions appearing in the graph could be selected by regular expression matches against their names.

The concordance generator proved extremely useful as an aid to studying the code. Its success was due, in retrospect, to several factors:

- The Web interface is hugely beneficial: it spares the user the effort of setting up the tool, learning how to invoke it, and dealing with the query syntax. It also eliminates all problems of platform dependence, because the tool runs on a single machine.
- By employing AT&T's dot graphing program (www.research. att.com/ sw/tools/graphviz/) as a back end, the tool could respond to queries with well-formatted, easy-to-read diagrams.
- The tool exploited NASA's rigorous coding standards: almost every function in the code had a standardized header with a brief specification and list of arguments, much in the style of JavaDoc, and these were brought out in the concordance enabling readers to rapidly learn the meaning of a function.

Message sequence chart generator

The same group that developed the function concordance also built a post-processor that converted message traces into message sequence charts, which show messages as horizontal lines between vertical lines that represent processes executing through time. This tool made it much easier to understand the protocols between the CM and the other components, which was a vital step in successfully replacing the CM.

Results

Our archaeological work on the existing CTAS system uncovered some previously unknown problems with the CM implementation, indicating that our reverse engineering reached below the surface level. A primary reason we could do this so quickly is the effective coding and commenting standards that NASA followed, which enabled Imagix and the concordance generator to support effective and rapid study of how various functions were implemented.

References

1. E. Kohler, M. Poletto, and D. Montgomery, "Evolving Software with an Application-Specific Language," *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS '99)*, ACM Press, New York, 1999, pp. 94–102.
2. T.J. Davis, K.J. Krzeczowski, and C. Bergh, "The Final Approach Spacing Tool," *Proc. 13th IFAC Symp. Automatic Control in Aerospace*, Pergamon, Oxford, UK, 1994, pp. 73–79.
3. R. O'Callahan and D. Jackson, "Lackwit: A Program Understanding Tool Based on Type Inference," *Proc. Int'l Conf. Software Engineering*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 338–348.

the Route Analyzer (RA) and Trajectory Synthesizer (TS), which collectively predict paths and arrival times for aircraft, and the Dynamic Planner (DP), which computes runway assignments and suggested delays.

In addition to its primary functions as message switch and database, the CM acts as the main process, responsible for initializing most of the other processes and terminating them when they appear to be behaving inappropriately. Neither FAST nor TMA can run without the CM, and if the CM dies, the system dies with it. In contrast, the system can tolerate failures of algorithmic or user interface processes.

Reasons for a Case Study on CTAS and CM

CTAS is an attractive subject for a case study for several reasons.

- It is a prime example of infrastructural software: software that operates vital systems such as transportation, medicine, power, and so forth. Working on CTAS lets us evaluate the viability of software engineering techniques for this important class of software systems.
- CTAS is large enough to present the complexities typical of large-scale software, rather than a small safety-critical

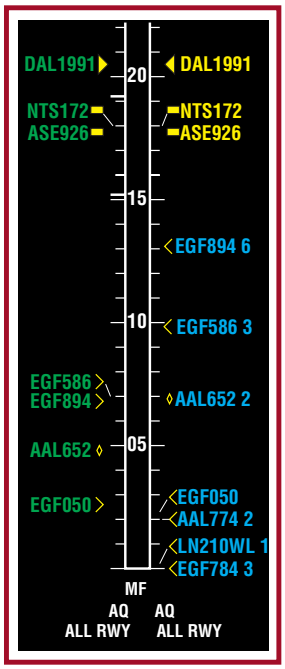

Figure 1. CTAS output, in timeline form. A marking on the central timeline of 10 indicates 10 minutes from the current time. The appearance of EGF586 on the left at 6.5 minutes from now indicates the time at which Air France 586 will arrive at the domain boundary if it follows its current trajectory. The appearance of "EGF586 3" on the right at about 9.5 is a suggestion to delay the aircraft by 3 minutes.

device. The code of CTAS is about half a million lines long.

- CTAS is not a strawman. Although most of NASA's efforts have been directed towards algorithmic aspects of CTAS, not its software engineering aspects, it is a well-constructed piece of software. Indeed, the US Federal Aviation Administration has officially adopted CTAS for nationwide deployment. As part of this effort, the FAA has hired Computer Sciences Corporation to extend CTAS with additional features. Even without CSC's extensions, the "research" version studied here is in daily use at DFW.

Within CTAS, this case study focused on the CM. While the algorithmic components are well understood and stable, the CM has grown steadily over the development cycle and has become the repository for various unrelated features, which were placed there only because they have no obvious home elsewhere in the system. As a result, the CM is complicated and its design is less elegant than the design of the rest of CTAS. It is also a single point of failure, so its reliability is particularly critical. Its source code is in about a hundred files and is about 80,000 lines long, of which about a quarter are comments.

The Existing Design

The existing CM design uses a functional decomposition. One module handles addition and deletion of flight plans, for example, another module interacts with the ISM, and a third maintains the assignment of aircraft to RAs. This design's control flow is implicit. There is a Motif user interface—not for the air traffic controller, but for system administrators to configure the CM itself. The main processing loop runs when called periodically by Motif.

Here is a partial list of problems identified in the existing design and addressed by the redesign.

- *Blocking sends*. The NASA developers regarded this as the most serious problem. Because the sending of messages uses blocking primitives, the CM could become deadlocked. At certain points, the CM would create more messages than the algorithmic processes could handle. It would fill its outgoing buffers and stall, waiting to write more. Meanwhile, an algorithmic process would send a message to the CM and stall if the CM did not process it. To mitigate this problem, the NASA team modified the CM to batch messages into groups of limited size. This causes considerable complexity in the current design and also makes the system's behavior as a whole hard to analyze.
- *Failures*. The system is not fault-tolerant. Although it can withstand the loss of an algorithmic process or a user interface, the CM is a single point of failure. If it crashes, the entire system must be rebooted. The FAA has specified that no system outage is to last longer than 25 seconds, yet it takes longer than that to restart the system and refill the aircraft database with fresh records.
- *Monitoring*. The FAA would like to add various monitoring features to CTAS that would let its behavior be continually evaluated, both to measure performance and detect symptoms of impending failure. Adding this to the existing CM is difficult, because it is not clear what the impact of inserted code would be, nor is it easy to find points in the code that should be instrumented.
- *Complexity*. The NASA developers are dissatisfied in general with the CM's complexity. Having never had the opportunity to redesign it, they have watched with concern as it has become increasingly complicated. The CM has become unwieldy, showing the properties of all software systems whose structure has degraded: small changes are hard to make and analyzing even simple properties of the CM as a whole is close to impossible.

We decided to focus our redesign effort on reducing the CM's complexity, convinced that many of the other problems would be ameliorated as a byproduct of this effort. Our design addressed the problems

of blocking and monitoring sends explicitly. We believe that other commercial systems might achieve the benefits we gained by focusing on complexity reduction, especially if they have evolved over a number of years.

We did not address the fault-tolerance issue as part of our group project, but one student designed an architecture that wraps and replicates the CM, demonstrating a successful implementation of this scheme by the end of the term. In our redesign, we also separated those parts of the state that cannot be trivially reconstructed on reboot from the rest, intending that these might eventually be stored in a persistent database.

The New Design

In our redesign, we investigated how the CM might look if NASA could redesign it from scratch. Our new design, shown in Figure 3, is dramatically simpler than the existing design. What surprised us was not just that we could simplify the design so extensively, but that we were able to do it using such standard and well-known techniques:

- *Data abstraction.* The existing design is built in a traditional, procedure-oriented style, in which procedures communicate by arguments and global variables that are bound to elaborate record structures defined in header files. Most of the redesign's components, in contrast, are abstract data types that encapsulate data structures and prevent direct access.
- *Infinite queues.* The redesign uses a standard message queue abstraction. By providing an illusion of an infinite queue with nonblocking reads and writes, it lets users write client code without any concerns for deadlock. We thus avoided the complexities that arose in the existing design from the need to avoid filling the outgoing buffers. This data abstraction is more intricate than the others. Unlike a traditional passive data abstraction, the infinite queue is active: it uses its own internal thread to move messages from the application-level virtual buffer to the limited-capacity operating system queue.
- *Generic message processor.* The existing design handles messages in a traditional style. A large case statement branches on the message type, and the message is then deconstructed and copied into a local record by type-specific code. The message-handling code is long and complicated, and there is much replication. In our redesign, a generic processor finds a handler appropriate to an incoming message by looking up its type and the type of the component that sent it in a table. Code that executes for all messages regardless of message type is factored out, so that each handler is simpler than a branch of the case statement in the existing system. Message handler registration is dynamic, making it easy to change the association between message types and handlers during execution. This simplifies monitoring: for example, incoming messages of a given type can be tracked by adding a new handler registered for that type.
- *Uniform external interfaces.* The existing CM has two input modes: in operation it receives input messages from the ISM component, while during testing it uses a different software subsystem to read input from a file. In our redesign, all input to the CM is via messages. To

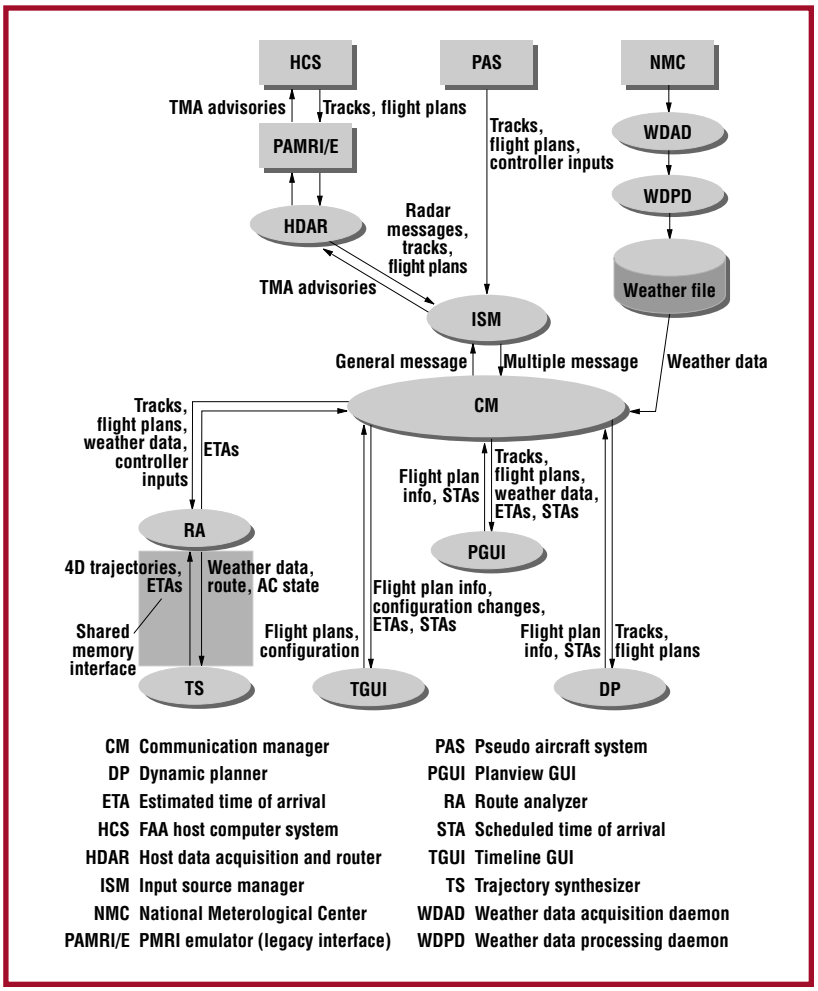

Figure 2. Architecture of CTAS in TMA mode.

The new design is not only simpler, but is more flexible, easier to analyze, and easier to tune. This is an example of Hoare's maxim that "inside every large program is a small program trying to get out."

run the system from the recorded data, we implemented a process that masquerades as the ISM, reading the file of recorded data and generating messages that are indistinguishable to the CM from real ISM messages. This scheme simplifies the CM and makes playback mode a better predictor of real behavior.

- *Message-handler language.* Rather than writing the message handlers by hand in Java, we chose to generate the handler code from a domain-specific message handling language. This language was designed to accommodate the existing C header file descriptions of message formats, so that we only had to write small code fragments to indicate, for example, how message fields should map to database records. (The design of the language and its compiler is orthogonal to the rest of the design.[1])

These techniques let us reimplement the primary CM functionality in 15,000 lines of Java, less than 20% of the previous amount of C++ code. This result is qualified, however, by several factors. We did not implement the administrative user interface used to configure CTAS. Also, we reused code from the existing CM in two areas: file parsing and message formats. The CM reads in the airport and airspace configuration from a file, from which it populates its internal data structures; we saw no benefit in rewriting this code, so we wrapped it in native Java methods. In our design, the recorded data file is parsed in a separate process, which we chose to code in C so that we could use existing parsing code. As we've discussed, our message-handler compiler (whose code size is not included in the line count above) also exploited existing C header files to generate code that extracts fields from messages.

Lessons Learned

Following are the opinions of the authors, and not necessarily the consensus of the class as a whole.

Simple designs are possible

The most obvious lesson is that a complex and successful software system can be dramatically simplified. The new design is not only simpler, but is more flexible, easier to analyze, and easier to tune. This is an example of Hoare's maxim that "inside every large program is a small program trying to get out."

Standard software engineering techniques work

Our project was not intended to demonstrate that standard techniques, such as data abstraction,[2,3] would solve the problem. In fact, we had hoped to experiment with more advanced techniques, in particular object modeling. But we brought about such major improvements using standard techniques alone that we never progressed to more ambitious ones.

In the last few years, the focus of debate about software development has moved from technology to process. More companies appear to be concerned with their organization's maturity level than their engineers' technical education. Our experience suggests that before considering refinements of process, it might be worth evaluating the potential of well-understood software engineering notions that have yet to be applied to the system at hand.

From an educational point of view, our experience suggests that undergraduate computer science courses should emphasize basic notions of modularity, specification, and data abstraction, and should not let these be displaced by more advanced topics, such as design patterns, object-oriented methods, concurrency, functional languages, and so on. In our experience, professional developers usually show much more sophistication in their use of algorithms than in their understanding of abstraction, which is usually more critical to a development's success. Perhaps this indicates that undergraduate teaching in algorithms is simply more effective than in software engineering. But it might suggest that our curricula toward issues of programming in the small, to the detriment of the needs of industry.

Coding standards are vital

Our reverse-engineering efforts benefited immeasurably from NASA's rigorous coding standards. Before embarking on a major development, it might be worth considering what kinds of tools might later be used to analyze the code, and how lexical and syntactic conventions might make their task easier and more productive. Bill Griswold has coined the

term *information transparency* to describe code that has been written with analysis in mind, and makes a compelling argument for paying attention to this aspect of design.[4]

Reverse-engineering tools work

Although we spent much time reading code, tools made a big difference. Their contribution was not so much in generating representations that could replace the code in reasoning about it, but rather in rapidly directing us to the relevant parts of the code pertinent to the question at hand.

High-level models are vital

Most of our reverse-engineering efforts focused on answering basic questions about the behavior of the system as a whole and the assumptions it makes about the aircraft behavior and airspace structure. CTAS's documentation, although several hundred pages long, does not include any system-level models that address these issues.

This is typical of industrial development environments. Such models are time-consuming and difficult to construct, and they can seem unnecessary to expert developers who have already formed them inside their heads. But the price paid for their omission is high. New developers who join a team acquire expertise in a slow and error-prone fashion, often from the code. Expert developers might have inconsistent models of the system, but, because these are not articulated, do not discover the inconsistencies until integration.

Most worryingly, we can miss the forest for the trees: the most basic and fundamental issues often get only scant attention because of pressure to meet deadlines and resolve low-level implementation problems. For example, the CTAS documentation does not explain the fundamental issue of how aircraft are related to identifiers. It does mention that, because the FAA computer can pass to CTAS both the record for the arrival of a flight and the one for its scheduled departure from the same airport, an aircraft cannot be identified by its call sign alone. This problem arose during a test at DFW and was fixed by appending the departure airport to the call sign. It is not obvious that even this is good enough, because if the FAA computer can send flight plans even before the flight has taken off, it might include flight plans for the same flight on two different days. In fact, the FAA's computer system holds at most one day's worth of flight plans, but this assumption is not documented anywhere. An upgrade to the FAA system that lets it store more flight plan information might thus cause a serious failure, and it is not hard to imagine the impact of such a change being overlooked.

Careful construction of an object model[5] would resolve issues of this sort early on in development, resulting in simpler, safer, and more economical software. An object model is simply one way to describe the abstract state of a system or its environment; what matters is not how the state is described but whether it is described at all. A report on the development of CDIS, an en route air traffic control system, corroborates this: it attributes only 8% of the total project effort to the task of constructing a specification of abstract states and operations, and claims that eliminating this phase would have increased the cost of the project overall.[6]

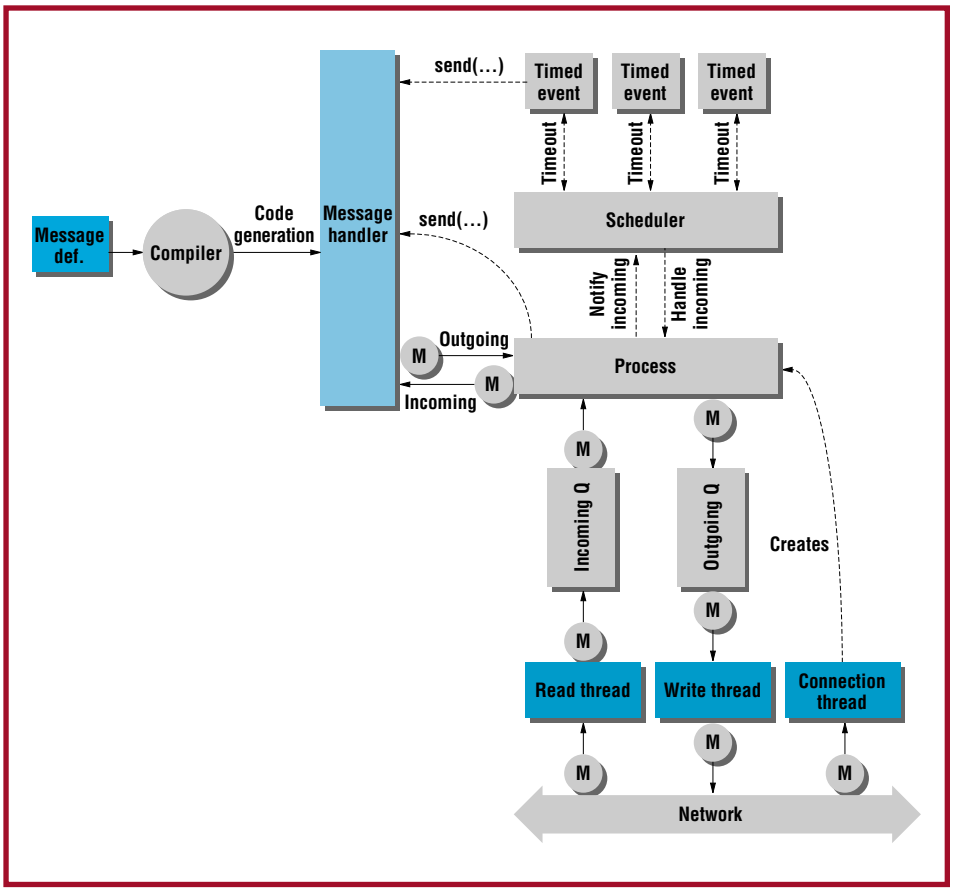

Figure 3. Informal architectural sketch of our new design of the CM.

The value of component specifications is now widely accepted. High-level models are perhaps even more important, because they are harder to extract from the code, and have a greater impact on the system as a whole. In addition, they are considerably cheaper to construct than precise component specifications.

The FAA is under pressure from airlines and air traffic controllers to field the current version of CTAS more widely as soon as possible. The delay that a redesign of CTAS would introduce cannot currently be justified, whatever the benefits. Whether the FAA is open to redesign in the long term is a different question. The FAA has reportedly set aside some tens of millions of dollars for the CSC effort to make CTAS more robust and add monitoring and control features. Our work demonstrates that a redesign of at least the CM is possible and would result in a smaller and simpler system, which could consequently be expected to be more robust.

More generally, our work supports an unpublished hypothesis articulated some years ago by Mahadev (Satya) Satyanarayanan of Carnegie Mellon University. He suggested that the common experience of deployed software systems both growing in size and degrading structurally as new features are added is an artifact of insufficient resource investment. A system to which sufficient attention is paid should actually shrink over time, as the developers improve their understanding of the problem and take advantage of more powerful tools. We observed this effect quite strongly in this case study. Our redesign benefited significantly from Java, a standard tool not available when CTAS development began, the message-handling script compiler, an application-specific tool developed after studying the CM's implementation, and a simple infinite queue data abstraction, which solved a problem whose importance was not recognized until the system had been implemented and its dynamic behavior became evident.

About the Authors

Daniel Jackson is an associate professor of computer science at the Massachusetts Institute of Technology, where he is the coleader of the Software Design Group with John Chapin and holds the Ross Career Development Chair in Software Technology. His research interests include all areas of software design, currently focusing on notations for design, tools for automatic analysis of designs, and tools for reverse engineering of code. He is a member of the IFIP Working Groups on Programming Methodology and on Software Requirements Engineering, and serves as associate editor of *ACM TOPLAS* and *TOSEM*. Contact him at MIT, Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA 02139; dnj@lcs.mit.edu; sdg.lcs.mit.edu/~dnj.

John Chapin is an assistant professor of computer science at the Massachusetts Institute of Technology, where he is the coleader of the Software Design Group with Daniel Jackson. His research interests include a wide range of design questions in computer science and software engineering, including operating systems design, multiprocessor memory systems, software engineering for parallel servers, and the foundations of parallel computation. Contact him at MIT, Lab. for Computer Science, 545 Technology Sq., Cambridge, MA 02139; jchapin@lcs.mit.edu; sdg.lcs.mit.edu/~jchapin.

Finally, our work underlines the power of software engineering fundamentals such as data abstraction, consistent coding style, and a design focus on simplicity. Both educators and industrial developers would do well to renew their focus on these well-understood techniques rather than letting them slide in favor of currently fashionable approaches.

Acknowledgments

Most of the reverse-engineering and implementation work was done by 11 students: Michelle Antonelli, Eric Bothwell, Chandrasekhar Boyapati, Tim Chien, Eddie Kohler, Charles Lee, SeungYong (Albert) Lee, David Montgomery, Massimiliano Poletto, Phil Sarin, Ilya Shlyakhter, and Tony Wen. Nadine Alameh and Mark Schaefer contributed in the reverse-engineering phase. In addition to the authors, one other faculty member, James Corbett (visiting from the University of Hawaii), contributed to the reverse engineering and to the design.

We are very grateful to our colleagues at NASA Ames for their help and encouragement: to Heinz Erzberger, CTAS's Chief Scientist at NASA, for his early enthusiasm that got the project going; to Michelle Eshow, the manager of the CTAS development, to Karen Tung Cate, her assistant lead, and to John Robinson, aerospace engineer, for their time in explaining CTAS to us, and for visiting us at MIT and reviewing our design work. Thanks also to Rick Lloyd and Ted Roe of Lincoln Laboratories for giving us their insights into CTAS and to John Hansman of the MIT Department of Aeronautics and Astronautics for teaching us some basic notions of air traffic control.

We thank Imagix Corp. and Grammatech Inc. for generously providing free licenses to the class for use of their reverse-engineering tools, John Blattner of Imagix for his helpful advice, Robert O'Callahan for help with his Lackwit tool, and Jean Foster and Alex Prengel in the faculty liaison office of Athena at MIT for setting up the class infrastructure.

References

1. E. Kohler, M. Poletto, and D. Montgomery, "Evolving Software with an Application-Specific Language," *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS '99)*, ACM Press, New York, 1999, pp. 94-102.
2. D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053–1058.
3. B. Liskov and J. Guttag, *Abstraction and Specification in Software Development*, MIT Press, Cambridge, Mass., 1986.
4. W.G. Griswold, *Coping with Software Change Using Information Transparency*, Tech. Report CS98-585, Dept. of Computer Science and Eng., Univ. of California, San Diego, 1998.
5. D. Jackson, *Alloy: A New Object Modelling Language*, Tech. Report 797, MIT Lab. for Computer Science, Cambridge, Mass. 1999.
6. A. Hall, "Using Formal Methods to Develop an ATC Information System," *IEEE Software*, Vol. 13, No. 2, 1996, pp. 66–76.