CHAPTER TWENTY-THREE

The HADS Team

Karl Rehmer

IN ANCIENT TIMES, THE BUILDER WHO WAS IN CHARGE OF BUILDING AN ARCH WOULD BE REQUIRED TO stand under the arch when the supports were first removed. If the arch failed, it would come tumbling down on the builder. I felt a bit like the arch builder the first time I got on a Boeing 777 airliner. I had written some critical parts of the flight code and been part of a team that built the software tools that were used to build a large portion of the flight software for the plane. It's an amazing feeling to realize that your life depends on the quality of the software you have written. As with building the arch, the final product was not the result of one person's efforts. The builder would know the history of the project, and sometimes knowing what was done could make him quite nervous. Though I knew a lot of the background of the development of the flight systems and some of the problems encountered during development, I didn't have to be nervous like the arch builder. The airplane had undergone extensive flight testing before it was ever put into service. I wasn't the first to stand under the arch. It is exciting to board a plane, knowing that your team played a big role in helping to produce the flight software. The story I'm telling here is the story of the HADS team, the team that built the compiler, runtime, linker, debugger, and other support tools used by the developers of the flight software for the Boeing 777. This small team developed, adapted, or maintained by the HADS team was roughly the same

amount of code as for the entire 777 flight systems that had hundreds of developers. Every one of the project deadlines was met. Working on this team was one of the highlights of my software engineering career.

I suppose some teams start off beautiful and stay that way. I know that some teams start off ugly and stay that way. The HADS team didn't start out as beautiful, and at the end, it just kind of faded away. But in the middle there, it was beautiful, and while it was beautiful, it was really beautiful.

The Background

You might wonder how Honeywell Air Transport Systems ever got involved with making the HADS tool set, since its purpose is to produce avionics for commercial aircraft.

The origin goes back to the 1980s. Sperry Flight Systems was making the transition from building analog flight controls to using digital computers and software for the controls. At one point in the process, Sperry designed its own computer chip and had a software team make a hybrid version of Pascal for this processor. They recognized that the type checking provided by Pascal provided some safety advantages and added some features that were important to writing software for flight systems. When the Ada language became a standard they became interested in the language for developing software for safety-critical flight systems.

In 1985, Sperry Flight Systems began looking at Ada for a fly-by-wire system for the proposed Boeing 7J7. The target processor was to be a member of the Intel 80x86 family with an Ada development system from a compiler vendor named DDC-I. DDC-I was respected as a compiler vendor, especially for the quality of its tools and runtimes for the Intel 80x86 family of processors. It also helped that DDC-I had a sales and engineering office in Phoenix, Arizona, where Sperry Flight Systems was located.

As part of the software development process for flight-critical software, all routines included in the system must be well documented and have tests written to ensure code coverage. An effort was undertaken to document and test the DDC-I Ada runtime system to the level required for DO-178B, the FAA guideline for software. In the middle of this process, Sperry Flight Systems was sold to Honeywell. The part that dealt with large aircraft such as commercial jetliners was named Honeywell Air Transport Systems Division.

Meanwhile, my wife and I were working as assistant professors at Indiana-Purdue University at Fort Wayne, Indiana. In the summers we did software consulting at a Magnavox facility in Fort Wayne. Magnavox was an early adopter of Ada, working on the first really large Ada project, called AFATDS (Advanced Field Artillery Tactical Data System). In 1988, we decided that my wife would continue her education by working on a PhD in computer science and I would support the family by finding a job in industry.

My Ada experience matched Honeywell's needs and I was hired to be a member of a small team that was to reverse-engineer, document, and write coverage tests for the DDC-I Ada runtime system that was to be used for the 7J7 project. For speed, the runtime was writ-

ten in assembly language, so the first part of documenting the runtime was to reverse-engineer the code and write a higher-level pseudocode description of each of the algorithms. The purpose of each subprogram and when it would be called was also documented. This team got to the point of having the runtime documented when the 7J7 project was cancelled by Boeing.

Honeywell Air Transport Systems had a continued interest in Ada, and knew that it would have future Ada projects, so I began working on developing design and coding standards for Honeywell's use of Ada. I also developed an in-house training class for Ada. Both of these were important because few software developers at Honeywell had any Ada experience or experience using many of the software engineering principles that Ada was designed to support. In short, I was the in-house Ada guru.

It wasn't very long until Boeing began to develop its next airliner, the Boeing 777. Honeywell Air Transport Systems was awarded large portions of the software.

Among the software to be developed was a system called AIMS (Aircraft Information Management System). This system was to be written completely in Ada. Previous flight software would have one CPU per line replaceable unit (LRU). AIMS integrated the functionality that had been distributed into multiple LRUs into a single system. A single processor would run several applications. For software that was flight-critical, underlying software needed to guarantee that one application could not steal time from or corrupt the data of another process. Because multiple time-critical applications that shared information were to run on a single processor, AIMS required a powerful processor.

Weight on an aircraft is always important. Every pound that is flown requires fuel. As part of the weight savings, AIMS was to be passively cooled—no fan was to be used to provide for cooling.

The hardware engineers at Honeywell did an extensive study of the available processors and rejected the most popular Motorola and Intel chips as either consuming too much power (therefore being too hot to passively cool) or not having enough processing power to handle the job. The analysis and the fact that the hardware engineers were able to build some early prototypes led the engineers to require the AMD 29050 processor for AIMS for the 777. The host system for development was DEC VAX computers running VMS, and later, DEC VAX workstations.

This presented an interesting problem, because Honeywell had promised Boeing that it would do the AIMS software in Ada, but there were no Ada compilers available for the AMD 29050. In my role as Ada guru, I recommended that the choice of the 29050 should be revisited. Since the quality of the compiler and support tools would have a great impact on the development, I felt that the choice of a mature compilation system was vital. The hardware engineers insisted that only the AMD 29050 would meet their needs, so Honeywell needed to find a development system. They approached a number of Ada compiler vendors about producing a compiler and related tools, but for various reasons none of these worked out. So Honeywell decided to make its own compiler and development

tools. After all, it had made compilers before. It did realize that an Ada compiler and tools was a more complicated task than creating a Pascal compiler, so it formed a partnership with DDC-I. It would use the DDC-I frontend technology as the starting point. In addition, two DDC-I employees would work as consultants on the project. Several engineers from Honeywell's Software Tools section were also to work on the project. I was not one of the original members of the team. I was to continue my guru role. The product to be delivered was called HADS, the Honeywell Ada Development System.

DDC-I had just completed a project of making an Ada development system for the Intel I960 processor. Since it and the AMD 29050 both are RISC-like processors, the source code for that project was selected as the starting point for the project. An Ada compilation system involves much more than just a compiler that generates source code. It also requires a runtime system to manage tasking, exception handling, and storage management. Ada's tasking is a way, within the language, to allow for communication and synchronization of independent threads of control. Exception handling, while now common in languages like C++ and Java, was not a well-known concept at the time. Ada's storage management allows for the management of a general heap as well as other specialized heaps.

The compiler and linker are host tools, and DDC-I had used its DEC VAX native Ada compilation system as the tools for developing its Ada I960 compiler, so this was a natural choice for writing the HADS compiler and linker.

The runtime portion of the compilation system is code that runs on the target. Since the HADS compiler was not yet available for generating the needed code, the runtime was written in C and AM29050 assembler. The portions of the runtime that would be included in the safety-critical portions of final AIMS applications would need to be documented and tested to DO-178B standards.

Since I wasn't originally on the team, my recollections of the initial team are based on things I could overhear while I was sitting in my nearby cubicle, as well as things told to me later. Some developers' names I will be using are not the correct names. Some of this is because I truly don't recall the name of a person, and some to avoid any possible embarrassment.

The Initial Team

While sitting in my cubicle in the same area as the members of the HADS team, I was able to observe the initial work of the team. The start-up was anything but promising.

Tom and Dave were the DDC-I consultants working on the project; Tom had been the project leader for the DDC-I I960 project. Among other things, Dave had played a vital role in one of the most difficult parts of the code generation process: the task of making efficient use of the registers on the hardware. None of the Honeywell engineers had any experience with building compilers other than perhaps in a class in school. Tom was not initially the project leader for the HADS project, but was always closely consulted.

Aside from setting up initial schedules, the first major task was to analyze the differences between the Intel I960 and the AMD 29050 to try to understand what pieces of the previous project could be retained, what could be modified slightly, and what would need to be completely rewritten. Engineers from Honeywell were to do this work with consulting help from Tom and Dave. Chris was to analyze the architecture differences that affected linking. John was to look at the architectural impact on exception management. Cynthia looked at linking issues. Waleen looked at storage management issues. Ajit looked at issues related to Ada tasking. Ajit also looked into issues related to what is called the User Configurable Code, since it interfaces closely with the scheduling portion of Ada tasking. The engineers doing these analyses would be designers of these portions of the tool set. Implementation would use some additional Honeywell engineers including Dan, Fred, Willis, and Henry.

It's interesting to observe how differently people work. From my nearby cubicle, I could get a lot of impressions of how things were going. John didn't ask for much help, and mostly studied documents by himself. Waleen usually didn't go to Tom and Dave for help. Since we had worked on the 7J7 documentation project together, she would often come to me to ask a question or two. Chris spent a lot of time with Tom, but mostly he would moan about how impossible his task was. Tom was not entirely sympathetic and would not take over Chris's task, but he would try to point out to him some areas that needed to be looked at and to point him to the documents where the information could be found. Fred was one of those guys who has a lot of great ideas, but then doesn't seem to follow through with them. I can remember several times thinking how glad I was to not be on that team.

After a few weeks, John decided that he was not up to the task and asked to be moved to a different project. Because of my past experience documenting an Ada runtime, I was asked to take his place. In spite of my misgivings, and because it would not have been politically wise to refuse, I joined the team with my first responsibility being the exception management. The first meeting to go over the differences was not too far away at that time, but John had actually left behind some pretty good documentation. By the time of this first high-level design meeting, I was able to speak to the major architectural issues that would impact exception management. Still, the fact that I was given a late start on the project put this part at risk.

At this same meeting, Chris made his presentation on the architectural differences that would impact code generation. It was obvious to everyone at the meeting that he had done virtually nothing. He had clearly started with one of the DDC-I documents for the I960 and made some editing changes. Frequently, entire sections were about the I960 with nothing said about the AMD 29050. In some places, it looked like he had just used the editor to substitute "AMD 29050" for "I960" and statements made were clearly false. This was really bad, since generating code is obviously one of the most important parts of a compilation system. I guess I was too amazed by the presentation to remember much about the reaction at the meeting. I don't recall any direct criticism of Chris, but certainly

errors and omissions in his presentation were pointed out. The meeting didn't do anything to increase my confidence in the success of the project.

Dave later told me that as they were walking back to their cubicles, he and Tom were discussing how poorly Chris had performed. They couldn't believe the lack of progress. They decided to go to the section head for the Honeywell Software Tools group and have Chris removed from the project. Interestingly, Chris had gone straight from the meeting to ask to be removed from the project. Some other areas of analysis, while not quite this bad, seemed also to be in trouble. The first major deadline, being able to compile, link, and run a "Hello World" program, was in jeopardy. Many on the team were not pulling their weight.

Getting It Right

Something had to be done to get the project on track. The team was reorganized, and in spite of looming deadlines, made smaller. Chris was not replaced. Fred moved off to a different project. Willis and Henry moved off to a different 777 support role, working with the C compiler that was used for a small portion of the 777 code. Peter, an additional consultant from DDC-I, was added. Tom officially took over as project lead. The team divided the work not previously done, identifying the unique features of the processor that affect code generation and the runtime implementation. The frontend of the compiler could be used without change. Only a very small amount of the code generation and a minimal amount of the runtime are necessary to compile and run "Hello World". Though it was close, the first deadline was met.

Following this, some team members began the detailed design and implementation of the storage management, task management, and exception management of the runtimes. Others continued with more code generation.

Waleen did the design and implementation of storage management. At a high level, many of the algorithms could be similar to those used for the I960. The biggest challenge was that this was being written in assembly language, so a complete rewrite needed to be done. Waleen also had to learn the assembly language.

A low-level body of code called the User Configurable Code (UCC) allows for the writing of a very small portion of code to allow interfacing of the runtime with the actual hardware. Some examples of code in the UCC are low-level code for I/O, code to manipulate timers, and code to handle traps and interrupts. This was designed and implemented at this stage. Ajit worked on the UCC.

For the most part, the task-management algorithms could be used as they were from the I960. The challenges had to do not with the Ada tasking constructs, but with general handling of independent threads of control such as how to do a context switch and handle interrupts. Since handling interrupts and doing context switches are very close to the hardware handling that is done by the UCC, Ajit also worked on this portion.

I worked on the design and implementation of the exception management. The general form and purpose of the algorithms did not have to be changed, but the low-level implementation involved a lot of architectural differences. This portion of the code was written in C. A C compiler for the AMD 29050 was available from Metaware, but there was no source-level debugger. An instruction set simulator provided the ability to run programs and to do machine-level debugging. It was possible to have the compiler generate mixed source and assembly listings, so this was used to relate the source code to the generated machine code for debugging. The lack of source-level debugging can be cumbersome, but with practice, the listings and machine-level debugging provided an adequate development environment. As a side benefit, I was forced to learn quite a bit about the machine language. This was helpful later when I did some work on portions of the code generator.

Cynthia continued working on linker issues and the other members of the team areas of code generation for additional constructs.

As part of the standard for Ada (Ada 83), a suite of tests called the Ada Compiler Validation Capability (ACVC) was defined. In order to be validated, a compiler and associated runtime must pass all the applicable tests of the ACVC for its environment. After the first deadline of being able to run "Hello World", subsequent deadlines of passing increasing percentages of ACVC tests were defined.

One defined deadline was to pass 80% of the non-tasking ACVC tests. When this deadline was met, the first release of the system was made to the users for the 777 team. Some groups were far from ready to begin coding at this stage, but others were ready to do some prototyping. One particular group led by Joel was especially anxious since he had heard rumors of Ada inefficiencies. Though many of the so-called inefficiencies can be met by turning off the default, language-defined runtime checks after initial development, the group was concerned. Immediately upon getting the first release of the compiler, they did an analysis of the output of the produced machine language. They concluded that the performance of the compiler was not good enough. In spite of the fact that we told them that the emphasis on this release was not on performance, and that subsequent releases would have much more optimization, they began to look for alternatives. They didn't really have much choice since Honeywell had committed to using Ada and there were not other compilers. They decided that for much of their code they would use Ada's machine code insertion mechanism. When this is done, the user essentially writes assembly language and the compiler merely assembles it. This would technically meet the mandate to write the code in Ada, but give the desired performance. After about the fourth release of the compiler, the HADS team revisited the original code from this team and found that the compiler generated much more efficient and smaller code than the "assembly" version. This is not surprising, and was shown to be true in a number of published studies. A high-level language compiler, and in particular an Ada compiler, can keep track of and take advantage of more global knowledge than a programmer can keep in his head. The compiler developers are also generally more knowledgeable of architecture features that can be used. The HADS team had to continually "sell" the product.

The project had one great piece of good luck related to the AMD 29050 processor, which was the existence of a very good and complete simulator. This meant that the team could develop and test while the actual hardware was in the early prototype phase. It turned out that any code that would run on the actual hardware would run on the simulator as well. There were only a couple of timing-related areas where code that was not legal on the actual hardware would run on the simulator. The simulator was a big factor in letting the team stay on schedule.

Though some deadlines were close, the HADS team met all deadlines on the way to producing a compiler that passed 100% of the ACVC tests. After meeting this goal, the work was far from done. For example, Ada defines and allows for a number of implementation-dependent features. These include things like the ability to precisely describe the layout of data structures such as records and arrays. Features appropriate for the AMD 29050 and for the 777 project needed to be implemented. Also, the promised optimizations needed to be added. Since the UCC portions of the project were finished and the Ada tasking was working, Ajit left the project to work for one of the 777 project teams.

Dealing with User Issues

One of the major tasks that we had to undertake was education and support for the users of the tool set. Most were unfamiliar with the Ada programming language, few knew any details of the processor architecture, and the tool set was new to all of the users.

Aside from dealing with some of the myths about Ada, the HADS team had to struggle with a general lack of Ada knowledge among the 777 developers. Ada is a powerful language with a lot of features that make it ideal for safety-critical embedded systems. These days it would not be considered to be a complex language, but at the time, it was considered to be a very complicated language to learn. Many of the software engineering concepts that Ada was designed to assist were foreign to the majority of developers. The HADS team had to take on a role as informal instructors in the use of Ada. For example, Ada's strong type checking can lead to difficulties getting programs to compile. This is an intentional attempt, in the design of the language, to catch inconsistencies as early as possible (during compilation) rather than at runtime. But struggling users tend to blame their tools. We were often called to help out some user with a compilation problem. Since most of us on the HADS team were very familiar with Ada, this was not difficult, but time-consuming. It also could lead to some tense encounters.

I remember being asked to go help Frank who was having some difficulties getting his code to compile. I went over to his desk and we sat down to go over his code. He definitely was having some type-mismatch problems with the subscripts of an array and the elements of the array. So we struggled through getting the types straightened out. Frank was quite frustrated with the strong type checking. I think he came from the school that feels that anything should compile and then you debug the result. Finally we got his code to a state where it would compile, but it seemed overly complex. I said to him, "You know, there is a better way to do this," and told him about some Ada features that would make

the code clearer and probably generate more efficient code. Frank got all upset, feeling insulted that I was criticizing his code. I was taken aback since in the HADS project we did this all the time and the team members considered it helpful. Of course, the members of the HADS team knew that they were good and didn't feel threatened. Frank had just gone through a lot of frustration just to get something to compile, and was then told there was a better way.

Passing all of a test suite does not guarantee that a compiler is bug-free. The ACVC tests are a rigorous set of tests for the basic features of Ada, but combinations of features may not be tested and implementation dependencies are not included in the suite. Naturally, the HADS team had to deal with bugs in the releases. One thing that we did well was, when possible, to duplicate a bug with a small test which was then added to our own regression test suite that was run in addition to the ACVC tests. Producing such a test case can be difficult.

One time Paul came to the team with a bug report, and after looking at it, we agreed that this did seem to be a code generation problem. Unfortunately, Paul said that he could produce the bug only in a large, complex system and that whenever he would try to simplify it, the bug would go away. Clearly he had found some obscure corner case. Luckily, as part of Honeywell, we had no difficulties getting permission to get all of Paul's code. By turning on some output of the internal workings of the compiler, we were able to see what the compiler was doing and we were able to reduce the original case consisting of several thousand lines of code down to a test program of about twenty lines. Sometimes it was not as easy as this, and we might spend several days creating a small test case. Still, the effort was worthwhile since finding a correction for a bug in a small program is much easier than trying to find the correction in a large program. The small program also provided the basis for the regression test.

I've often said that software tools are at the bottom of the food chain. By that I mean that when all else fails, users blame their tools. We had to deal with this issue a great deal, since not only was there a tool to blame, but it was a "homegrown" tool, and therefore was more mistrusted. One day, I got a support call from Frances who was complaining about a compiler bug. When she was writing to an array, erroneous code generation was causing another nearby variable to be overwritten. This kind of behavior is possible, perhaps even likely, when using a language like C that doesn't check that an array subscript is in the proper range. With Ada's compile and runtime checks, this behavior should be caught. If it is statically known at compile time that a subscript will be out of bounds, a warning will be emitted at compile time. At runtime, checks will raise an exception if an attempt to access an array with an out-of-bounds subscript is done. Such runtime checks do generate extra code and can create inefficiencies, so Ada compilers allow an option to turn off some or all runtime checks. Frances had done her build using an inherited build script that turned off runtime checks. A small amount of investigation and debugging showed that the source code was written so that values were written beyond the end of the array. The compiler was generating code to do exactly what it was instructed to do. We had to continually educate users on the tools' command-line options and their effects.

We had to deal with many bug reports that turned out to be user error. Unofficially we called these UIB (User Is Bozo) errors.

Once the ACVC tests were passing and several releases with some implementation-dependent features were made, the team began working on creating a source-level debugging environment for the HADS system. About this time, Cynthia was taken from the team to work on the 777 project. The remaining members of the team did the initial work on the debugger. Tom, Dave, Peter, Dan, and Waleen then returned to compiler issues while I finished the first release of the debugger. Tom concentrated on peephole optimizations, Dave concentrated on efficient use of registers, and Dan and Waleen concentrated on general choice of optimal machine code generation for a construct.

One of the things that Dave needed to do in providing for efficient use of registers was "live/dead" analysis to determine when a register is holding a variable and when that variable is no longer used. As a side effect of doing this analysis, it was easy to determine whether a register was being read before it was written to. This would mean that an uninitialized variable was being used. Dave generated code that would emit a warning about an uninitialized variable being used. Once this was released to users and they began to get the warning, they began to call, asking for help to find their uninitialized variables. Dave found a clever way to analyze the inputs to the register allocation whereby he could output the name of the uninitialized variable. He implemented this and wrote several small test cases that showed that it worked properly. We included this feature in the next release. Almost immediately, we started to get complaints that the compiler had slowed down, and as we looked into it, we found that it was this uninitialized variable analysis. For one system that had compiled in under five minutes with the previous release, we aborted its compile after a day and a half. Naturally, the compiler was not any slower when there were no uninitialized variables, but it was clear that the ability to name any uninitialized variable detected was not worth the expense.

Adding information about the uninitialized variables was just one example of a request for a feature that came from our users. Since we were in-house and since many of the users had gotten to know us, we were often individually approached to add a feature to one of the tools. This began to get out of hand and we soon adopted a policy of "just say no" to any such request. We referred all of the requestors to our project lead, Tom, who would decide, along with the rest of the team, whether the request should be added to our work and what priority it should have.

We also instituted a "top 10" list for each person. Each person had a list of his top 10 priorities. That way, he knew what he should be working on now and what was coming up next. Tom might revise these lists daily, but if we disagreed with the priorities, he was open to discussion. We also had weekly meetings, on Friday afternoons, where we discussed the progress of the previous week and helped set priorities for the coming week. A small amount of technical discussion might also occur at these meetings.

When we were making the source-level debugger, much could be taken from previous DDC-I work, but many areas of debugging are architecture-specific. The handling of debug

information is architecture-specific, and each target has specific ways of doing things such as setting breakpoints. Because the simulator was so useful to the team, a debugger that supported both the simulator and actual targets was made. As with the compiler, the release of the debugger was not a "big bang"—rather, a series of releases, supporting more and more features, was planned.

With the release of the debugger, more user support was required. One issue that was particularly perplexing was variables in registers. The code ported from the DDC-I baseline did a good job of indicating the initial location of a variable in a register, and where the register was stored if a subprogram was called while in the scope of the variable. But when we started to add optimizations to reuse registers, we found that we did not have information about how a variable might be moved from the register into memory and the register given to another variable. Users would ask for the value of a variable at a point where it was dead and its register reused. They would get an incorrect, confusing value when really the correct answer was that the variable was no longer available at that point in the code. It was a significant task to design a system to track this and to implement it in the debugger and in the register assignment portion of the compiler.

Many users are unfamiliar with using source-level debuggers, having mostly used print statements to get information about their programs. Educating users about how to use the debugger was an ongoing task. It seemed that no one reads manuals. As users would get familiar with the debugger, they would often ask for the addition of a feature that would make debugging easier for them. Usually it was easy to "fill" these requests since the feature was already in the debugger and the user hadn't read the manual.

Even with source-level debugging, sometimes it is useful for users to be able to debug at a machine code level to examine the low-level behavior of the code. Of course, when debugging at this machine code level it is useful to know something about the architecture of the machine and its assembly language. Unfortunately, many users didn't bother to read that documentation either. One day, Sam called me up and said, "Either this debugger is broken or there is something wrong with the simulator." So I went over to his cubicle to see whether I could find the problem so that we could fix it. He was debugging at the machine code level where he could step individual machine instructions. He said, "Look, here I am ready to execute a call statement, but when I step into the machine code, I end up at the statement after the call rather than in the called routine." It was hard not to laugh. The architecture utilizes a "delay slot" to keep the instruction pipeline full when doing a branch or call. So the behavior that Sam was seeing was the correct behavior.

A similar experience came when Helen called up to tell us that the compiler had optimized an important statement of her code. She was stepping through the code at the machine code level and got to a call statement. The source window was reflecting the source code positions that corresponded to the machine instructions executed. She said, "See, the assignment that is done just before the call was not done." Our optimization had moved the assignment into the delay slot. If she had executed one more machine code step, she might have been surprised that she didn't immediately get to the called routine, but she would have seen her assignment being done.

Epilogue

Developing a large tool set like HADS can be an expensive proposition. In addition to acquiring rights to the frontend compiler technology, external consultants and Honeywell engineers were used on the project. While this was not a large number of people, it still had a cost. Some of the managers on the project seemed to resent the expenditures. At one point in the project, an Ada compiler for the AMD 29050 from an external source became available. Since there was not much of a potential market for a compiler for this target, it is likely that the target customer for this compiler was the 777 project. It may be that some at Honeywell were encouraging the compiler vendor to develop the tool set to replace HADS or as a fallback position if the HADS team should fail. One day in a hallway, Bruce, one of the managers, came up to me and said, "Since there is an off-the-shelf compiler available, they are going to kill the HADS project." Naturally this was concerning, not for the sake of my job as there was plenty of work to be done, but because we took pride in the quality of what we had done and didn't want to see it go to waste. But Bruce was known as a guy who liked to spread a lot of rumors and who liked to make others uncomfortable, so we didn't do anything special. We just continued on with our plans. By the time the other compiler came out, the HADS team had demonstrated numerous successes. Some evaluation of the other compiler was done, but the HADS compilation system was clearly superior.

The small HADS team was very successful, producing a large, quality system without missing any of its deadlines. Of course, we had some advantages that the 777 team did not have. For example, there is a standard for what it means to be an Ada compiler, meaning that the basic requirements for the product were known from day one of the project. Another big advantage we had was that we did not need to produce the volumes of customer documentation that the 777 project's waterfall life cycle required. Except for documenting those parts of the runtime that required DO-178B certification, we produced only the internal documentation it deemed necessary for reference and only external user documentation. This made things quite a bit easier for the HADS team than for the 777 team.

But there were also some things about the team that led to its success. What were the things that turned an initially ugly team into a beautiful, successful team?

The most obvious change was that a number of people who were merely going through the motions were replaced. The result was a smaller but much more dependable group. Having people that you can't depend on is not good. With a larger team, someone who is not "pulling his weight" can be overlooked for quite a while. With a small team, it is obvious what each person is doing. We found it much better to have a small group of dependable people. The small group that did the majority of the work was also highly competent. Many of us had not worked on compilers and related tools, but we all had worked on difficult projects and had developed a great deal of competence in programming in general. Experience in developing compilers did not seem to be highly important, but experience that involved proper use of software engineering principles did seem to be important.

It was much more than dependability and competence. A collegial attitude also developed. We all shared several characteristics that led to success. Most importantly, we all knew we were good. Because we knew we were good, we didn't worry about taking credit or getting blame for individual accomplishments. Because we knew we were good, we would ask others for help when faced with a particularly nasty problem. Because we knew we were good, we did not take offense if someone else offered a possibly better solution to a problem. Because we knew we were good, we were willing to take time to help out others. Because we knew we were good, we recognized the competence of our colleagues. Because we knew we were good, we didn't feel that it was necessary to get our own way, but could be insistent when it was important. Some have said that HADS was a very egoless team, but I think that the team members all had pretty big egos; however, the big egos did not need to be fed. We all had a confidence in our abilities and the abilities of our colleagues.

We all worked very hard, and each of us worked to the maximum of his or her abilities. We worked very little overtime, but each of us came in on time and put in a hard, full day every day. Because we would ask others for help when we had problems, and because we could see our colleagues working hard each day, we always had confidence that the work was progressing.

During most of the development, each developer had the primary responsibility for developing a particular area. Some example areas were the storage manager, for the exception manager, providing for linking that matched the AIMS requirements for separation of applications, register assignment, and optimizations. Since we knew the person primarily responsible for an area, it was easy to talk with the right person about an issue.

In addition to the competence and personalities of the team members, there were several practices that the team used that led to success.

Since the HADS project was being developed in-house, and the engineers on the 777 project got to know the HADS developers well, it was easy for the project engineers to approach one of us about a particular feature or optimization that he desired. This could have led to a lot of "feature creep" or conflicting requirements. We avoided this problem by insisting that all requests go through Tom, the team leader. He decided what requests should be worked on, as well as the priority of these requests. He did a good job of shielding the rest of the team from a lot of outside pressures. I'm sure that it was a difficult balancing act to protect the team and also be responsive to the 777 project.

A really helpful practice we used was to have a short (half-hour) meeting at the end of the day on Fridays. In this meeting, we assessed the progress of the past week and set the general priorities for the next week. This was pretty much a democratic process, though Tom, as team lead, had the final say. Very little general problem solving was done in these meetings, but we were allowed to give opinions about things to look into as a possible solution to a problem. Many agile teams currently use a similar concept, though they have shorter, daily meetings.

We had deadlines that were rigid on data and fairly rigid on content. The weekly meetings determined the primary focus for the next week. The "top 10" list allowed us to continually readjust focus onto the short-term problems that must be solved in order to achieve the long-term goal. The combination of longer-term deadlines, goals for the next week, and the ability to change daily through the "top 10" list provided for an agile development environment.

The development of a regression test suite, based on issues found either internally by the team or by users, provided continual feedback on the quality of the product. It also made it easy to find regressions introduced during development. When a regression was found, its seriousness could be evaluated to determine its priority and when (and where) it should be added to someone's "top 10" list.

The HADS project was quite enjoyable to work on. Much of this had to do with the beauty of the team and the success of the project. It is difficult to know whether the success, and the practices that led to success, caused the team to be beautiful, or whether the beauty of the team led to the successes. I suspect that they grew together.

Any project winds down as it either fails or produces what is needed. For the HADS team, the wind-down was gradual. Several members of the team were moved to areas of production for the 777 project where their expertise in Ada and their intimate knowledge of the tool set were beneficial. Gradually, the consultants were removed until the team was down to two people. In a final round of layoffs, I was laid off and transferred to the Honeywell Technology Center. After a few months, the one remaining team member grew tired of working by himself and he left Honeywell. Maintenance of the project was taken over by a completely different set of engineers. An irony of the project is that after all of the primary HADS developers had left the team, the HADS project was given Honeywell's Highest Technical Award.

The perspectives given in this narrative are my own. Others on the team may perceive things quite differently. Perhaps, though I doubt it, some of the primary developers would not feel that the team was beautiful. But for me, my experiences on the HADS team were one of the highlights of my career as a software engineer.