Ship System 2000, a Stable Architecture under Continuous Evolution

Björn Källberg, Rei Stråhle

SaabTech Systems AB
S:t Olofsgatan 9 A
SE-753 21 Uppsala
Sweden

rest@systems.saab.se

1. ABSTRACT

Ship System 2000 was conceived in the middle of the 1980s. It was started as a development project for a general Command and Control and Weapon Control Systems for naval ships (C2 and WCS), intended for reuse. Its first use was for three different ship types. Following these orders, later deliveries with similar functionality were made. From the beginning the company chose to use the Ada language for future systems in this area and selected Rational as the first programming environment. A strategic decision was made to include all the Ada components developed in a family of reusable components. The number of components has grown and variants for different platforms and compilers have been included. In the early days the word was *Reuse*, and later the terms *Dual Life Cycle* and *Product Line Management* have been introduced for the same concept. Since then, the 1st generation has been successfully reused for ships from many other navies and air forces. However, the architecture has evolved in two major steps, and several minor steps. Thus, it is still a modern architecture, but with a heritage.

SIGAda 2001 09/01 Bloomington, MN, USA
ACM 1-58113-439-8/01/0009…$5.00

2. SYSTEM DESIGN

The system is a distributed system, where typically 20 nodes are connected using a double Local Area Network (LAN), for redundancy reasons. Each node is a single or multi-CPU node. In principle there is one node for each sensor, weapon and operator. From the start, we tried to use as much commercially available equipment and software as possible. The first generation used M68K series computers, and the Ethernet LAN was based on the ISO standard (i.e. not IP). At that time, (the end of 1980s,) the coming dominance of IP was not easy to foresee. The operating system was OS9, which is mostly used for industrial automation purposes. The networking system was mostly custom made by us.

The latest version is based on Windows NT on the Intel architecture, using IP on top of FDDI (a 100 MHz fibre network, with standardised redundancy and excellent fallover characteristics, especially compared to Ethernet). Software-wise, the system consists of some hundred different software components, CSCs. Each component typically consists of many Ada packages, with ten to a hundred thousand lines of code. All of the application components are written in Ada, but a number of the lower level components are written in C.

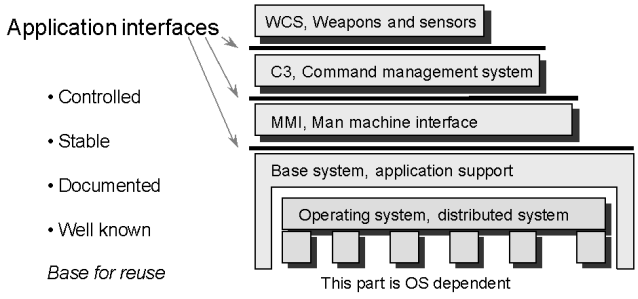

Figure 1: Layered structure

The components are organised in a layered structure, which hides the operating system and other implementation issues from the application

programmer. The reuse degree between different systems is very high, typically 80-90%. In the base system and application support, the reuse is 100%, but in the higher, more application-oriented layers the reuse is lower. As an example, if a system has a new type of radar that has not been used previously, a completely new component is written for that sensor.

The unit of distribution is an Ada program. Each program consists of many CSCs, linked together. The programs send messages to each other over the LAN.

The fundamental design decision of the components is that all interfaces are software interfaces, defined by Ada packages. Thus, the actual format of a message transmitted over the LAN is completely hidden within one component. It is never the case that a message is transmitted by one component and received by another component. Instead, those components essentially consist of two parts, a client package that is linked together with other components, and a server side, which receives the data transmitted by the client package. See Figure 2 below.

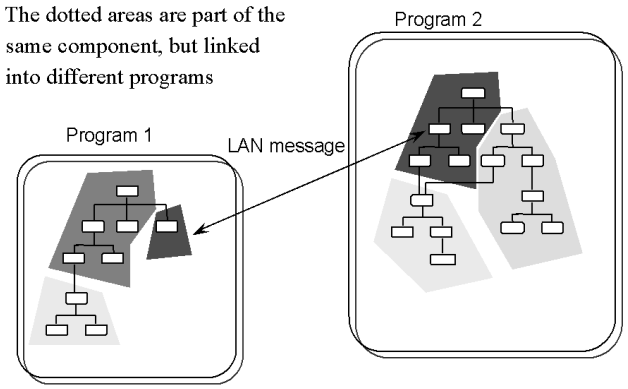

Figure 2: Ada program structure

3. APPLICATION INTERFACE STANDARD

A number of guideline documents for writing applications was developed very early in the development of the base system. This set of instructions defines the fundamental rules for a component. A component must follow these rules to be integrated and work within the architectural rules of the system.

The fundamental areas covered are the following:

- **System Start and Redundancy.** The system will automatically reconfigure in case of problems. This requires special uniform start up and shut down procedures.
- **Error Handling Principles.** Naturally, error handling is performed uniformly over the system.
- **Overload Behaviour.** Programs shall be designed to minimize effects of overload and always cause a graceful degradation.
- **Parameterisation.** Rules the process of parameterisation of software in the system.
- **Inter-Program Communication.** Principles for communication between programs.
- **Man-Machine Interface (MMI).** All communication with the operator is done using a special subsystem that isolates the program from the actual I/O.
- **Task Priorities.** How to set priorities. Essentially we use the Rate Monotonic Scheduling algorithm.
- **Input and Output.** Input and output to devices that are not MMI devices.
- **Time.** How time is implemented. This chapter was specially included to handle monotonic time in Ada83.
- **Operating Modes, System and Local.** Used to handle different modes, i.e. simulation and live.

4. EDUCATION / TRAINING

Very early on it was clear that education in the principles of writing code for the base system was needed. Also training in selecting from the existing components and their proper use was defined.

Since 1991 there have been 15 formal training workshops, including about 12 – 15 students each. Typical length of such a workshop is some 7 – 10 days with a mixture of both lectures and programming. Some basic concepts like the inter-program communication and the error handling principles are included, as well as areas closer to the application like coordinate conversions and track distribution.

The following topics are normally covered:

- Base system overview
- Inter program communication
- Error handling principles
- Program start and stop
- Base types and coordinate conversions
- Track distribution
- System parameters
- Man-machine interface

5. Ada95 TRANSITION

When the Ada95 standard was released, some worries arose in the company that the existing code, written in ANSI Ada83 / ISO Ada87 over 10 years by a great number of different programmers, needed to be modified to a large extent in order to conform to the new standard. A first conversion was made with a relatively small system, some 700k lines of code.

The results of this first conversion were very promising:

- Of the 6 new keywords, only one was used in 1 place.
- The transition to 8-bit character set required changes in 3 places, where an "array of character" was used.
- New type Wide_Character made concatenation of string literals ambiguous in 10 places.
- Generic parameters can in Ada95 not be instantiated with unlimited types, unless a specific syntax is used. This required changes in 4 places.
- Changed attributes for Float, 10 changes.

In the following cases deviations from the company programming standards were detected and the code was in some sense unsafe or erroneous which was detected by the Ada95-compiler.

- In Ada95 the parameters for the pragma are checked and are in some sense valid. This was found to be erroneous in 6 places.
- The fact that Numeric_Error and Constraint_Error are the same in Ada95 required changes in 3 places.
- In Ada95 it can be determined by the package specification if a package body is required or not, it is never optional. That required changes in 20 places.

Hence a total of 60 changes were required. Since the code in the study encompassed 735 846 lines of Ada source code (comments not included), it can be estimated that the backward compatibility is 99,99%. In Ada95 there is a standardised pragma Interface, which differed between the compilers earlier used in the company. If this had been utilised, changes would have been required at about 100 more places.

It is now an accepted fact that we use Ada95 compilers to develop our systems with a minimum of local design rules and rely heavily on the Ada95 Quality and Style.

6. UNEXPECTED EXPERIENCES

In this chapter some difficulties (actually both expected and unexpected) are highlighted.

6.1 Documentation

During development we followed a typical military development model, which requires thorough documentation for each component and subsystem.

We had hoped to be able to reuse almost all of the documentation between the different projects. However, this was not possible to the extent we had hoped. The reason was security problems. The requirement specifications for each system is of course classified, although they mostly contain similar requirements. The problem is shown by the following example:

Assume customer A has the requirement that 800 tracks shall be presented every second, but customer B requires 1500 tracks per second. The requirement for the reusable component must then be that it shall handle 1500 tracks per second. This is written in the SRS, Software requirement specification for the component. However, this SRS can not be shown to customer A, because he then can deduce the secret requirement of customer B. Thus, different SRS had to be produced for the same component.

Similar problems occurred for other documents, but at least some could be reused.

6.2 Dead Code

The customer does not necessarily see reuse of code as an advantage. The following disadvantages can be seen. Reuse almost always means, that a general component is used, which does not exactly match the specific customer need. As customers have specific requirements, it means that a component that shall handle requirements from multiple customers must be the union of all requirements. Thus, the component will contain code that does not correspond to the requirement for that specific customer. This code can occur in many forms, with different forms of disadvantages:

- Extra functionality which can be used. This makes the system larger, more difficult to learn for the operators. Of course, in some cases this may also be an advantage, the customer gets useful extra functionality for free.
- Executing code with functionality that can not be used due to absence of accompanying MMI. Uses system resources.
- Code that is loaded, but will not be executed - uses memory.
- Code that is not loaded, but is part of the total system anyway - uses disk space.

As the code is larger than strictly necessary, maintenance, if done by the customer, will be more difficult.

6.3 Complexity

An important part of the system design process is to divide the components in such a way that the union of the requirements is not too much different from the requirements from the individual customer. If this is not successful, the complexity of the system will grow beyond control.

Assume we have three different systems, each consisting of 4 CSCs. The requirements for each CSC, and the complexity for the CSC, had it been made specifically for that project, varies according to the table below. Assume further, that the complexity of the system is the product of the complexity of the individual components.

In the simple system below, the complexity of the reusable system is than 25 times as complex as the

most complex delivered system. Obviously this is not a good component design.

Thus, a component must not be too general. If the requirements are very diversified, it is better to make a new, simple component for that specific case.

We have used both ways, both general components, where the functional adaptation was made by setting parameters controlling the behaviour of the component, and having different components. Early in the development program, we could probably have benefited by separating more components.

	Project A	Project B	Project C	Reusable system
CSC 1	1	7	8	8
CSC 2	4	10	2	10
CSC 3	5	1	9	9
CSC 4	10	1	2	10
Total complexity	200	70	288	7200

Table 1: Complexity for individual projects, compared to system complexity

Our reuse effort has been successful. One of the reasons probably is that the application domain is clearly defined, and limited. For larger, more general domains, the complexity problems will be more difficult, or even impossible, to handle.

7. PORTABILITY

The system has been ported to many different environments, operating systems, and Ada compilers. The latest is the port to Windows NT.

The transfer of code to a new compiler is always difficult. We use a rather complicated structure of nested generics. Thus, no compiler has ever compiled our code the first time, it has always required some bug fixes to the compiler.

Porting to a new operating system is relatively straightforward, due to the layered structure of the system.

8. PORTING TO WINDOWS NT

Porting to a new windowing system was a more exciting challenge. In the older system, we used a proprietary windowing system, with a limited number of windows. The challenge was to port this to Windows NT, and getting the look and feel of a modern windowing system. This was possible due to the MMI structure of our system. The application programs never directly access the MMI, but read and write all MMI data to a real time database. There are then different MMI programs, that also reads and writes from the same database. In this way, the applications are completely decoupled from the actual layout and implementation of the MMI, see Figure 3.

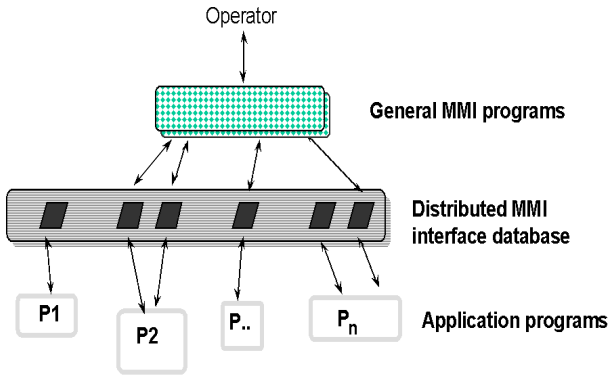

Figure 3: MMI architecture

In the first generation, all general MMI programs were proprietary, written in Ada. In the NT version, about 50% of the Ada programs are kept, in modified version. These modified Ada programs now send messages to Visual C++ programs, which handle the direct operation with the operator. In this way, most of the application logic is still kept in Ada, but we can utilise the power of the GUI tools that come with Visual C++. All application programs are unmodified.

9. EVOLUTION

Looking back at more than 15 years of development in this product line, a number of important evolutionary steps can be recognised. The first major step was a conversion to Unix, IBM AIX, and a different application domain, an intercept control system for the Swedish Air force. It also includes a large simulator for the same system.

The second major step is a sibling to the first, and involves a conversion to Windows NT. Some of the architectural aspects that have made this continuous evolution possible are:

- Layered structure

- Fine grained structure, not monolithic
- Openness to other architectures, both hardware and software
- Location independency
- Asynchronous messages
- Parameterised components
- MMI definition language
- COTS Operating system
- Ada

The different steps have not only involved a change of operating system, but also other major changes such as:

- Use of commercially available windowing system (X-windows and Windows NT) instead of proprietary systems
- Change of network protocol
- Replacement of custom made bit slice boards with general purpose computers
- Porting between different Ada compilers
- Coexistence of different languages (Ada for the applications, C and C++ for lower level functions and window handling)

The majority of the application components are unaffected by these changes. The components have however been maintained and upgraded over the years, by many different programmers. We attribute this to a large degree to the use of Ada.

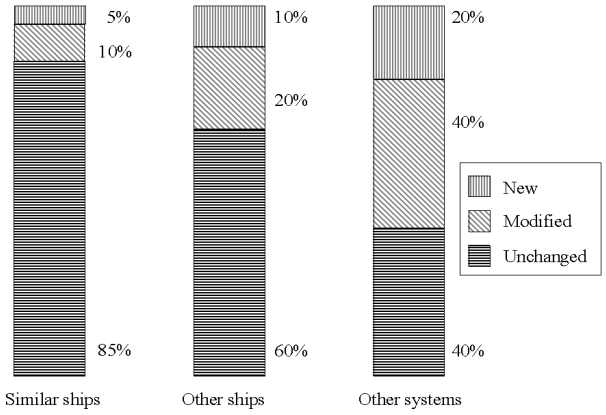

Figure 4: Degree of reuse

10. CONCLUSIONS

Some of the experiences we have drawn from the continuous development are that reuse is not easy to accomplish. This architecture has been able to evolve with different requirements, different platforms and different people over a time period of more than 15 years. We attribute a large degree of the success to the fact that we chose to use the Ada language from the very beginning.

There is however a great need to use different ways to control the increase in complexity that can be introduced by too general components, and perhaps the most important factor is that reuse should be applied to a limited application domain.

Documentation can not be underestimated. It must be complete and relevant in order to bridge the information gap between the developer and the user of a component. And often many years have passed from the time of development until the reuse. Remember that reuse is not necessarily seen as an advantage by the customers, but can still be very successful, if applied with a large degree of skill through the analysis, design and coding phases.

11. REFERENCE

Bass, Len; Clements, Paul; Kazman, Rick: "Software Architecture in Practice", Addison-Wesley, 1998.