Interviews on Software Systems Merge

Rikard Land

Mälardalen University, Department of Computer Science and Electronics
PO Box 883, SE-721 23 Västerås, Sweden
+46 21 10 70 35

rikard.land@mdh.se, http://www.idt.mdh.se/~rld

Abstract

As a follow-up to previous research on software in-house integration, we have studied a case where integration has been achieved by means of developing and reusing components of the system from previously two separate systems. As an important part of this research, a number of interviews with people from an industrial project have been carried out. The present report describes the methodology used to set up these interviews and contains the full notes made during the interviews. No analysis of the results is made.

Keywords
Software Evolution, Software Integration, Software In-house Integration, Software Merge, Software Systems Merge.

1. Introduction

Within an organization, from time to time it is realized that two or more in-house developed software systems address similar needs, and there is an overlap in functionality. This may be caused by gradual evolution and extensions to systems that initially addressed different needs; this is also typical when the organization changes through new types of collaborations and mergers. The in-house controlled software may be products of the companies, or some support systems for the core business. In either case, there is typically a desire to rationalize maintenance as well as to business considerations to reduce the number of similar systems offered to customers and users. To study how this situation has been addressed in industry, we have previously performed a qualitative multiple case study. The main data source were interviews, and the details of that study, including the copied out interview notes, was published as a technical report [12]. Several analyses has later been carried out based on this study [5,8-10,13].

One result of the study was that there are several high-level strategies, which might include retiring some existing systems and evolve others, or start developing a new generation and plan for retiring the existing ones [5,9]. The most complex and unknown strategy was labeled *Merge*, and means that components are reused from the existing systems and reassembled into a new system [5,9]. This strategy has been little researched and is not well understood, and is little performed in practice (as far as we have seen). A main difficulty involve the different kinds of incompatibilities between components from different systems [5,9]. To study the topic of software systems merge, we designed the present study, which can be seen as a follow-up of one of the previous cases, where a *Merge* is currently being implemented.

1.1 Research Questions

The study concerns mainly technology, and the assumed context is that any incompatibility analyses has to be performed mainly at a high level, to early in the process be able to make a well-founded decision how to merge [13] – if it is practically possible at all. Also important for success is that the merge will often need to be evolutionary [5,9], meaning that some stepwise deliveries must be made of the existing systems where they share more and more common parts [13]. Given this context, the two main research questions are:

Research Question 1. How can important incompatibilities be characterized?
Research Question 2. What can be done to solve these incompatibilities?

The envisioned context of the method is the small group of architects who typically meet and outline various solutions [11,13]. Therefore, the compatibilities should be characterized and discussed at a high level. Although the software architecture community mostly focuses on architecture as a structure of components [1,15,17], we are also interested in other high-level aspects that we assume would make merge more or less difficult, such as the data models of the systems [5-7,9], and the technological frameworks chosen (in the sense "environment defining components") [10]. One focus of the study is the potential difference between functionality that is encapsulated into components or modules, and such that is scattered throughout the code, i.e. so-called cross-cutting concerns (modeled as "aspects" in the aspect-oriented community [3,4]), which could be one way of characterizing Brooks' notion of conceptual integrity [2]. Another focus is how choices of common parts affect the rest of the existing systems, and under what circumstances the shared component should be modified or the surrounding parts in the existing systems.

1.2 Acknowledgements

Many thanks to the interviewees and their organization for sharing their time and experiences. Also many thanks to Laurens Blankers, Jan Carlson, Ivica Crnkovic, and Stig Larsson for our previous and future collaborations in this research on in-house integration and merge.

1.3 Disposition

Section 2 describes the background of the case, and section 3 presents the research method in depth; data collection methods and how threats are addressed. Appendix 1 contains the discussion agenda of the interviews, and Appendix 2 collects all interview notes. There are no analysis, summary or conclusions sections, as the purpose of the present report is to make available the raw data (the interview responses) and describe the research methodology in detail. Analyses of the material is planned to follow in separate publications.

2. The Case

The organization is a US-based global company that acquired a slightly smaller global company in the same business domain, based in Sweden. To support the core business, physics computer simulations are conducted. Central for many simulations are two 3D simulators, developed separately by the previously two separate companies. Both consist of several hundreds of thousands lines of code and implement advanced state-of-the art physics models. The number of staff maintaining and evolving these simulator systems is three for the US system and two for the Swedish system. There was a strategic decision to integrate or merge the systems in the long term. This should be done through cooperation whenever possible, rather than as a separate up-front project.

In previous publications [5,8-10,12,13], this case was labeled "F2" (which then also included related programs that were used to pre-process input data for the 3D simulators).

The interviewees describe both systems as a main loop, invoking certain physics modules at appropriate times. During program startup, an initialization module reads data from file and initializes the data structures in memory, and after the simulation is finished, there is a module that collects and prints data to various files. Both sites had experienced problems with the same type of physics model, and wanted to improve it significantly (independent of the merge). The starting point of integration discussions where therefore this particular module (that we can call "PX" for "Physics X", there are also other types of physics modules involved, that we can call "PY", "PZ", etc.). As a consequence some other parts and aspects of the systems also needed careful consideration: an error handling and logging module ("EL"), the shared data structures in memory ("DS") and a file management module ("FM"). See Figure 1.

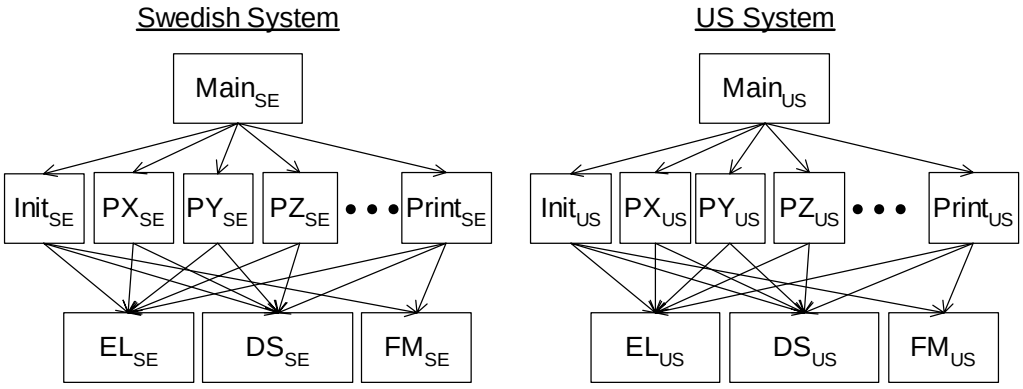

Figure 1: The structure of the existing systems (a module view).

3. Research Method

The research strategy chosen is that of a case study [16,19], although it also bears resemblance of "grounded theory" research [16,18] due to the exploratory nature of the research.

3.1 Data Collection

Data was mainly collected through interviews, although we also were given high-level documentation of the Swedish system. For the interviews, we used an open-ended discussion agenda (included as Appendix 1). Robson gives some useful advices on how to carry out interviews in order to e.g. not asking leading questions [16], which we have tried to follow. We contacted the same people as during the previously mentioned multiple case study, i.e. the developers of the Swedish system, and also were helped with contacting the developers of the US system.

The five interviewees are all developers currently involved in maintaining and evolving the two systems. As we are focusing on the technology aspect of the problem, we did not interview managers or project line managers

The interviewees have been given the following codified names:

- **SE1.** The main maintainer of the Swedish system since a number of years; physicist by education, but a long experience in software development.
- **SE2.** The main physicist behind the Swedish system during the last years with long experience in software development; writes much of the new code.
- **US1.** The main maintainer of the US system since some seven years; physicist and software developer.
- **US2.** The technical expert of the US system, responsible for the physics; does no actual software development. (Confirms interviewee US1 and adds a few details.)
- **US3.** Involved during the last two years in making the US system work together with new common things. (Confirms interviewee US1 and adds a few details.)

As the discussions (and the documentation) clearly revealed the same overall picture of the systems' structures (as described in section 2), the interview notes have been organized according to this structure (included as Appendix 1). (However, not all interviewees have been involved in all parts, so some slots in this structure may be empty for each individual.)

3.2 Addressing Threats

There are many threats to a sound research design. These are often categorized into construct validity, internal validity, external validity, and reliability ([19], p.34). The threats are discussed below, together with an account for the measures taken to address them.

3.2.1 Construct Validity

Construct validity concerns ensuring that the objects being measured and the measures used really highlight the concepts being studied. Yin gives three advices ([19], p.34), which are followed as described below:

- **Use multiple sources of evidence.** We are using several interviews as well as some documentation.
- **Establish chain of evidence.** The exploratory nature of the research makes this advice somewhat less applicable, at least until data analysis and reporting. During these phases however, it will be made sure that conclusions are traceable to data, and that all data is considered in the conclusions.
- **Have key informants review draft case study report.** This will be done, by sending the analysis to the interviewees asking for feedback.

3.2.2 Internal Validity

With internal validity is meant making sure any data contributing to the conclusions are properly assessed and documented, so that e.g. not spurious relationships are mistaken for true causes and effects. Three threats to internal validity are [14,16]:

- **Description.** It is essential that the descriptions of data (e.g. interview notes) are accurate. In case of the present research, this is addressed through "member checking", i.e. letting the interviewees review the copied out interview notes [16].
- **Interpretation.** Being open-minded and allow the data to explain themselves without imposing a framework upon them is essential, although it can be argued that this is impossible in practice – a researcher's previous experience will certainly be an important element during the interpretation. As the present research aims at building theory, this threat is somewhat less applicable.
- **Theory.** Alternative theories that may explain the same events should be considered. This is also somewhat less applicable to the current, theory-building research phase.

Triangulation is an important means in general to achieve internal validity [16,19]. Data triangulation means that multiple sources of evidence are used; in the case of the present research this means interviewing several people and using documentation as well. Also, letting interviewees check the interview notes is a valuable means to avoid researcher bias and increase internal validity [16,19].

3.2.3 External Validity

External validity concerns establishing the domain to which a study's findings can be generalized. As we have a single case study, generalization will have to be build on argumentation. To a certain extent, some cases of the previous multiple case study can be used to broaden the study's data base.

3.2.4 Reliability

Reliability concerns the repeatability of the study, and involves establishing and reporting many practical data collection issues. Any one studying the same case should be able to repeat the data collection procedure and arrive at the same results (at least unto some point of subjective interpretation). Yin's two advices ([19], p.34) and their application to the present research is described below:

- **Use case study protocol.** The case study protocol can be described as a workflow:
  1. Formulating research questions and discussion agenda.
  2. Booking and carrying out the interview. Taking interview notes.
  3. Copying out the notes (sometimes between the same day and three weeks afterwards).
  4. Sending the copied out notes to the interviewees for review. This has several purposes: first, to correct anything that was misunderstood. Secondly, to consent to their publication as part of a technical report, considering confidentiality. Third, some issues worth elaborating may be discovered during the copy-out process, and some direct questions will then typically sent along with the actual notes. These thus reviewed, modified and approved notes are used as the basis for further analysis.

- **Develop case study database.** All notes are kept in a binder for future reference. All copied out interview notes are stored in a CVS system.

4. References

[1] Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.

[2] Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition* (20th Anniversary edition), ISBN 0201835959, Addison-Wesley Longman, 1995.

[3] Kiczales G. and others, Aspect-Oriented Programming, *ACM Computing Surveys*, volume 28, 1996.

[4] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J.-M., and Irwin J., "Aspect Oriented Programming" , In *Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, pp. 220-242, Springer-Verlag, 1997.

[5] Land R., Blankers L., Larsson S., and Crnkovic I., "Software Systems In-House Integration Strategies: Merge or Retire - Experiences from Industry", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, 2005.

[6] Land R. and Crnkovic I., "Software Systems Integration and Architectural Analysis - A Case Study", In *Proceedings of International Conference on Software Maintenance (ICSM)*, IEEE, 2003.

[7] Land R. and Crnkovic I., "Existing Approaches to Software Integration – and a Challenge for the Future", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, Linköping University, 2004.

[8] Land R., Crnkovic I., and Larsson S., "Concretizing the Vision of a Future Integrated System – Experiences from Industry", In *Proceedings of 27th International Conference Information Technology Interfaces (ITI)*, IEEE, 2005.

[9] Land R., Crnkovic I., Larsson S., and Blankers L., "Architectural Concerns When Selecting an In-House Integration Strategy - Experiences from Industry", In *Proceedings of 5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, IEEE, 2005.

[10] Land R., Crnkovic I., Larsson S., and Blankers L., "Architectural Reuse in Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of First International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2005.

[11] Land R., Crnkovic I., and Wallin C., "Integration of Software Systems - Process Challenges", In *Proceedings of Euromicro Conference*, 2003.

[12] Land R., Larsson S., and Crnkovic I., *Interviews on Software Integration*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-177/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2005.

[13] Land R., Larsson S., and Crnkovic I., "Processes Patterns for Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI)*, 2005.

[14] Maxwell Joseph A., "Understanding and validity in qualitative research", In *Harvard Educational Review*, volume 62, issue 3, pp. 279-300, 1992.

[15] Perry D. E. and Wolf A. L., "Foundations for the study of software architecture", In *ACM SIGSOFT Software Engineering Notes*, volume 17, issue 4, pp. 40-52, 1992.

[16] Robson C., *Real World Research* (2nd edition), ISBN 0-631-21305-8, Blackwell Publishers, 2002.

[17] Shaw M. and Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, ISBN 0-13-182957-2, Prentice-Hall, 1996.

[18] Strauss A. and Corbin J. M., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (2nd edition), ISBN 0803959400, Sage Publications, 1998.

[19] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

Appendix 1: Discussion Agenda

Structure

Describe the structure of the system. What components are there? What are their roles? How are they connected; how is data and control transferred? (Is there any documentation of this?)

Framework

How are components defined? Do you utilize e.g. any language or operating system constructs? To what extent can modularity be enforced? To what extent do you rely on conventions (e.g. different files/directories with standardized names)?

Conceptual Integrity

For each of the following concepts *X*:

a) Error handling
b) Physics PX
c) Data structures
d) More?

Describe:

1. The *X* component

- Today and in the future:
  - Functionality
  - Interface
- How did you define the future component, in terms of the existing POLCA/ANC *X* component? To what extent did you try to achieve some similarity with today, and to what extent did you try to create something as good as possible?
- What will it take to move from today's component to the future *X* component?
  - How difficult will it be to modify the system to always use the new *X* component?
  - Did you assess this explicitly?

2. Any rules associated with *X* (which the whole system must follow):

- Today and in the future:
  - What does *X* required from the rest of the system?
  - What does *X* prohibit?
  - What would happen if these rules are not followed? Would the run-time behaviour be unpredictable/error prone, and/or would the system becomes more difficult to maintain?
  - Are these rules documented? Are they known? Do you enforce these rules in any way?
- How did you define the future rules, in terms of the existing POLCA/ANC *X* rules? To what extent did you try to achieve some similarity with today, and to what extent did you try to create something as good as possible?
- In terms of rules, what will it take to move from today 's system to the future *X*?
  - How difficult will it be to modify the system to always use the new *X*?
  - Did you assess this explicitly?

Appendix 2: Interview Notes

In this appendix, the complete notes from each interview are reported. In each case, the notes were edited by the present author to consist of complete sentences etc., and in some cases to remove details. The resulting text was then reviewed by the interviewee, and with minor modifications from them, the result is reported here.

Interview SE1

Structure

There is a main loop calling the physics modules at appropriate times, approximately once per iteration. Each module accesses the common data structures, and also makes use of the common error handling.

Framework

Current

Previously, we have relied on conventions of file names and directory structure, and talked about "modules" in the sense "code with associated functionality". We have used Fortran "Common" blocks (untyped memory areas) for global data structures.

Future

We are moving towards the new constructs available in Fortran 90/95. The "Module" construct supports the separation of functionality, and also makes it possible to statically check that the correct number and types of arguments are passed to a routine (in older Fortran versions, the source code files were compiled in any order and then linked, ensuring only that the names of the routines being called were valid). The "Type" construct enables us to create large data structures in memory with names and types for each element.

Components and Conceptual Integrity

The parts that are (or will be) common are: data structures, error handling, the physics module, and the storage mechanism. Ask interviewee SE2 for the status of these new parts and to what extent they are tested and being used.

Data Structures

Current Data Structures

Component. There are today Fortran "Common" blocks (untyped areas) with all the data.

Rules for usage. The data structures are initialized early from input data. There is no way to restrict access to different parts of the data. There is no type support from the environment, so the programmer must know how to treat every address – as a real, an integer, etc.

Future Data Structures

Component. We will use Fortran 90/95 "Types", which makes it possible to access the global data by name, and we will also have type rules.

Reasoning. As we needed to agree on a common data structure, we wanted to take the opportunity to also use of the more mature constructs of Fortran 90/95. The new structures are defined and needed by the new physics module, but this requires additional work in the other modules. Modifying them should be straight-forward though, if the structures are complete

Status. Ask interviewee SE2.

Physics Module

Current module

Component. There is a big fat subroutine implementing a large number of physics equations. It is one of the more model-heavy modules. It is called once per iteration in the program, and the data being processed is passed via vectors. There is a smaller subroutine that implements the core of the physics.

Rules for usage. The large subroutine is called only from one place (the main loop). The smaller subroutines are sometimes called from other places as well. How well-defined? The data is read and written in a complex way, which is due to the complex physics model.

Future module

For details, talk to interviewee SE2. The new module implements a new physics model, and the old physics module will exist in parallel for a while, both for validation purposes and to avoid forcing users and customers to switch to the new model.

Error Handling and Logging

Current Error Handling and Logging

Component. There is a routine initiated early during program execution. Later, there is a routine being called with a few parameters: the name of the subroutine where the error occurred, an array of strings (messages), and a severity flag (warning/error/stop execution). Within this routine, the messages are printed to screen and output files.

Rules for usage. Wherever the programmer suspects that an error may occur, tests are to be written, ensuring e.g. that data is within bounds. These bounds are hard-wired and have been adapted over the years. Sometimes, an error is ignored and an error code is sent upwards from a routine, and the caller needs to decide whether this is an error or not – this is done fairly systematically. There are some convergence tests in the main loop, to ensure that the overall simulation is not running amok. In the case of numeric errors (e.g. floating point exception) the program is terminated by the operating system and an error message is being written.

Future Error Handling and Logging

Component. For the new component, ask interviewee SE2.

Rules for usage. In future versions of Fortran, there will be stronger support for catching some errors. The code will then check the status of some global variable after a statement is executed (instead of trying to detect before a statement is executed whether it may produce an error).

Reasoning. We need a common library to be able to write common code. I see no problems with having two error handling systems at the same time, so that all newly developed common parts will use the new error handling component, while the old parts will use each program's respective older libraries.

Status. Ask interviewee SE2.

Storage

Current Storage

Component. There is a bottom layer of library routines for writing to flat files. On top of that we have routines for writing a bunch of words, and on top of that routines for writing the large data structures. There is also some simple data in the summary output file. The program also produces a text file with both some text data and some plots (with ASCII graphics).

Rules for usage. The data files are read at the beginning of program execution, and the data structures are saved once per program iteration.

Future Storage

Component. The changes will occur at a fairly low level, to enable an improved file format. If we change anything at the higher levels, that would be the result of things we want to improve at the same time (such as the current mix of integers and reals in the same array).

Rules for usage. No change.

Interview SE2

Structure

At a very high level, there is a main routine which nicely calls some other routines in a well-defined order, which read data from central data areas in memory (that are initialized early during program execution). First, some initialization routines are executed, then some iterations of the (several) physics modules, then some post-processing, cleanup and output.

Current

But at this second level there are many problems. With one word: spaghetti. The program is not well modularized, there are data dependencies between modules through the use of Fortran "Common" blocks (untyped global data areas in memory). There is no control on which parts of the program may read and write different data items. There are very large complex routines, that should be split into smaller, which cause many problems. There are also copies of the same code with only small modifications, implemented as routines with similar names but for slightly different purposes. All this results in a very complex logic, difficult to follow, although it should be simple; it is difficult to see from the source code which of these alternative routines that will actually be called.

Future

The high level structure (main routine, different physics modules, common data structures etc.) will be as it is, although all the unnecessarily complex exceptions and almost-identical routines will be cleaned up.

Framework

Current

The program is written in Fortran, and is continuously being updated towards Fortran 90/95. In Fortran 77 there were no mechanisms aiding in modularizing the code, and no constructs for error handling. So, everything relies on conventions, often not documented but practiced.

Future

The program needs to be (and will be) better modularized, with cleaner separation between parts. We are moving towards Fortran 90/95 and the new implemented parts are using the Fortran 95 construct called "Module". This makes it possible to package routines and data explicitly together, declaring them as public and private, per method. Unfortunately, the whole data structures must be declared as either, there is no lower level of granularity for data.

Components and Conceptual Integrity

The starting point was that a new physics model was needed for one particular kind of physics, both in the US 3D simulator and the Swedish 3D simulator. Due to management pressure, there was initially no resources allocated to build the common fundaments that are needed. After running into problems, a common error (and logging) package and approach was defined and implemented, as well as common data structures. A common storage (file formats and libraries for accessing these files) is also under development, that is in the long term intended to be used by all simulation programs within the company.

Data Structures

Current Data Structures

Component. There are large global areas where data is written to and read from. It is implemented as Fortran "Common" blocks, i.e. untyped areas. This means that the program (and the programmer) must keep track of how to interpret a particular address – as an integer, or a real, or… there are even arrays with mixed reals and integers, this is very old-fashioned style (belongs to the sixties) and is almost criminal.

Rules for usage. As the program has grown so complex, it is very difficult to understand which module writes data to which data areas, and when. It should not need to be like this. Only a small subset of the data structures with physics data needs to be shared; most of the data is only read and written by a single physics module (plus the initialization and printing routines of course).

Future Data Structures

Component. We are implementing the data structures as typed structures using the Fortran "Type" construct, where different variables can have names and types, and can embed other own-defined structures. As far as possible we are keeping the data private to the module and provide "put" and "get" routines, which may contain some logic to prohibit overwriting a certain variable under certain circumstances. However, for performance reasons we allow direct access to arrays, as it would be too costly to copy large arrays via parameters. Unfortunately there is a limitation in Fortran 90/95 so that the whole data structure must be either public or private, which means that we have to open up direct access to more data than we would like to. This limitation will disappear in future Fortran versions.

Rules for usage. As we know which physics modules need which data, we have designed the structures so that not more than needed is shared. Only the other modules that need to access the data should be able to do so, and in particular modifying data should be restricted to the modules that really need to, which we know from the physics models. This is not formally specified or enforced however.

Reasoning. We wanted to get rid of the many poor practices currently used in both the existing programs (US and Swedish). We therefore did not look at what it looked like in the existing systems. It is the same physics data being used, and consolidating the existing data modeled was not so difficult – it took a week of meetings, where we designed (and implemented) the structures (at the same time; since the Fortran "Type" construct is a very declarative way of programming, it functions both as specification and implementation).

Status. The US system uses the new data structures (because they use the new physics module). They have had to modify other parts to use the new data structures, but they have not fully replaced all of their previous data structures. They have also possibly had to add some small variable that we did not think of.

Physics Module

Current module

Component. The physics module implementing this particular physics model is not modularized. It does not use the newer Fortran construct of "Module". It consists of one large routine containing (unnecessarily) complex logic and programming. It is too long and complex and is dangerous to modify. It should be split into a number of smaller routines. The communication between the different smaller routines that are used is unclear – there is a mix of parameters, common data areas, and reading from input data files. It is one of the most important parts of the program, but also one of the worst. Some of the smaller routines it calls are shared with other parts of the programs.

Rules for usage. The large main routine of this module is called from the main loop of the program at appropriate times.

Future module

Component. The future physics module will be properly encapsulated, using the Fortran "Module" construct. This module contains only one entry routine, that others may call. There are a few parameters, for setting some calculation options but not for data – data is transferred and accessed through the new data structures.

Rules for usage. Similar as before: the large main routine of this module is called from the main loop of the program at appropriate times. We have restricted access to the smaller routines.

Reasoning. We did not look at what it looked like in the existing systems. It is being called with an entry routine, but everything under this could be new. The new module is not only new software that will replace the existing software, but also implements a new physics model. However, it was realized that to be able to create a common module, the fundaments needed to be common as well: the data structures and the error handling.

Status. The US system uses the new module and the new data structures. They have had to modify other parts to use the new data structures, but they have not fully replaced all of their previous data structures. Here in Sweden, I have encapsulated the old module, and modularized it into a number of smaller routines (called by the only public entry routine). It will be co-exist in parallel with the new module for a while, for validation purposes. It works well but is not fully tested and is not yet included in the release version of the program. I have not yet modified it to use the new data structures, but this should not be very difficult.

Error Handling and Logging

Current Error Handling and Logging

Rules for usage. During program initialization, the package is initialized, requiring e.g. some environment variables. In the source code of the program, the programmer must add appropriate checks if data is missing, if data is out of (reasonable) bounds. We rely on the common sense of the programmers to perform the checks needed in each given situation, but it is essential that the programmer is educated in numerical methods (we have some bad experience where this was not the case).

Future Error Handling and Logging

Component. There are a few methods: “write message” (writing an output message) and “add message” (adding a message to a stack, which might later be dumped if some serious error occurs). There is also a parameter indicating the seriousness of the error –error, warning, or only a message. These methods are used both for writing error messages and other logging messages. The possible messages are defined in a translation file from constants (used in the code) to strings (that appear in the output files). The messages may define slots for including variables retrieved during runtime of different types, and there is a way to automatically define new messages with a new set of variables. This allows for a central overview and management of possible messages, and enables easy addition of new (natural) languages. The output files where errors and messages are written are centrally managed, via environment variables and input.

Rules for usage. Same approach as today: during program initialization, the package is initialized, we rely on the programmers common sense to do the necessary checks.

Reasoning. When I sent my code (physics module) to the US, they had to replace all calls to the error package (still our previous). Of course they should change it to using their own error package, but often these calls only was commented away. This was clearly unacceptable from a quality perspective, so we needed to change the way we worked to avoid this type of double work resulting in lower quality. It was decided that the proper solution was to create a common error package.

Status. The new error package is used by the new physics module, which is used in the US program.

Storage

New storage is currently being developed by others, and will be common for other programs as well.

Current Storage

Component. We are today dumping the complete memory, which is a bad thing to do – one should save name/value pairs. The files are not possible to read outside the program, are not portable, and we have problems with backward compatibility.

Future Storage

Component. I do not know what it will look like.

Rules for usage. I do not know.

Reasoning. We want to have a common package across several programs as well, and get rid of some of the problems with portability, backward compatibility and understandability.

Status. There are currently discussions and initial design of this parts, made by others.

Other Remarks

We still do not have a common CVS repository US/Sweden. We have to synchronize/merge our changes when we meet (approximately twice per year).

The communication between parts is the fundament and must be done first! In our case, communication is in the form of common data structures.

It happens that one of us (US or Sweden) needs to improve a certain aspect of the program. Often the response of the other side is "we would like that too, but we currently have more urgent things to do". It is not possible to wait for the other side to be in the same phase, so one side invents and improves things in a way so that the other side cannot use it. And later, when the turn has come to the other side, there is no motivation to spend time changing something they have just improved, just to make it more similar. The motivation for developing common parts cannot come from within the scope of either programs, but must come from higher management as a strategic decision that will pay off in the longer term.

I think we will never reach to a point where we have a common program. This is mostly due to the fact that we perform two different types of calculations. Large parts of the calculations are identical, but some early parts of the calculations and initializations will differ, and I doubt the benefit of packaging these differences into a single program.

Interview US1

History

The US system is some 200 thousands of lines of code (KLOC) and contains eight large physics modules. The US system's earliest ancestry can be traced back to the early sixties. The first release under the current name was in the eighties. I started working with the system some seven years ago.

After a previous merger, the current system was able to replace other programs completely, but it can currently not replace the Swedish system as we only support one of the two main types of calculations done. Neither can the Swedish program replace ours. We are therefore evolving both systems in a common direction, with more and more parts in common.

The changes described have been implemented, but not yet formally released.

Structure

There are approximately eight large physics modules, which contain smaller modules as well. The logic calling these modules is fairly simple: first do this, then do that. There are large data structures in memory, used by the physics modules. The calculation starts with some initialization (from input files), followed by the physics calculations, and ending with some printout tasks (to output files).

90% of the code is identical for both main types of calculations done.

The current overall structure is difficult to understand, modify, and maintain.

Framework

We are moving from older versions of Fortran to Fortran 95 and newer. It contains things similar to object oriented practices: structures of data ("Type" keyword), interfaces with the "Module" keyword, which leads to more modularized code. We are getting rid of Fortran Common blocks, and now split data according to whether a variable should be local to a module, or visible from several modules.

Components and Conceptual Integrity

The parts that are common are: data structures, error handling library (although we do not use the new library yet), one physics module, and the data files library.

Data Structures

Previous Data Structures

Component. We have used Fortran Common blocks for storing the data in memory. In both the US and the Swedish system, the data structures were too old.

Rules for usage. The data in the common blocks are accessed by each subroutine through including the common block files (include "xxxx.h"). In this case, all data in the common block always comes together with the fixed order. It is an error precursor and hard to maintain.

Future Data Structures

Component. We have defined structured data using the Fortran "Type" construct, together with the Swedish project team. Doing this took about a week, including going through the existing data in both systems.

All the existing physics modules have been modified to utilize the new data structures. This took ca six months, which I consider is quite good. It had to be done, either now or later. It would cost more to postpone this work.

Rules for usage. We have defined system-wide conventions such as how a coordinate is defined: row/column or column/row.

Reasoning and Status. We have not modified the input and printout modules to use the new structures, because we want to wait with this and also modify the output format and the logics structure within these modules, to make it more flexible. To also change the input and printout modules to use the new data structures would cost three months each. We chose to have a translation from the old data format to the new (performed after init but before physics calculations), and from the new to the old (performed after physics calculations, before printout). It took one month to implement the translations. The more important reason to take translations is the old input/output process will be completely replaced by new modules in the next development phrase.

Physics Module

Previous module

Component. The old module works well in most operation condition. But it needs to generate data file in each cycle design and takes quite a long time. Also the old module faces more and more difficulties in handling more and more complicated situation.

Rules for usage. The old module is mainly called in the feedback loop.

Future module

Component. Even if we did not have the integration with Sweden, we would definitely replace our current physics model with something similar to the Swedish model. Sweden had experienced some shortcomings in their module, some things that existed in our previous module. However, it was not so urgent for them to implement a new model, but it is an improvement over the previous. The accuracy required is different depending on which type of calculation is being done. We need more accurate results so we could not use the Swedish module directly but need to use more data in the calculations.

Rules for usage. At a high level, the module is used as the previous components in both systems.

Reasoning. When defining the new model, it was very important to think about the data structures, as the quality of the code is highly dependent on the data structures! The decisions led to each other: the starting point was that we wanted a new physics module for this kind of physics, which led to definition of common data structures, reasoning about input/output formats, error handling, etc.

Status. The new module is used in the system and will be part of the next release.

Error Handling and Logging

Previous Error Handling and Logging

Component & Rules for usage. The existing error handling component is basically only a simple routine, to be called each time an error occurs. Errors are caught by test some variables before a statement is executed.

Future Error Handling and Logging

Component & Rules for usage. The new component is developed in Sweden, and works similar to the existing library in the US system, but with more powerful feature.

Reasoning & Status. The newly developed physics module is developed in Sweden and uses a new error handling library. We have not switched to the new one yet, and this far I have had to comment out all calls to it. I add some calls to our existing error handling module. This is of course not efficient. The next thing will be to install the new error handling module and start using it in all modules, as we want to standardize the error handling and logging, and utilize more powerful features. We will however not spend effort on changing error handler in the init and printout modules, but having two modules in parallel in the program would not cause a problem for the user.

Storage

Previous Storage

Component. Previous storage were user's input (text) and binary files.

Rules for usage. One subroutine with library to save or read all necessary data to/from input data files (i.e. data that is not user input).

Future Storage

Component. We now use common input data files (i.e. data that is not user input). The common module to handle these files is newly developed.

Rules for usage. Data is read just before the calculation when it is needed.

Reasoning. We wanted to change storage to a portable, standardized format. It is almost impossible to transfer old data to new one.

Status. The Swedish storage is used in the system and will be part of the next release.

Other Remarks

The simulation results from our system are very close to measured data.

In the beginning we only had local funding. Later, when we had a more clear picture of what was needed we got some central funding for (some of) the extra efforts due to integration. It appears as the management and funding system does not work very well. I see a danger that the systems will remain only partly common for a long time. Management is still interested in making the systems merge as much as possible. There are short-term costs but long-term benefits.

We use different development environments, in the sense source code control systems, and systems for make-ing the code. Previously we developed for different platforms, but we have currently switched to the same. For the common modules we also have common automatic tests.

Interview US2

Background

I am technical expert, the one who makes everything work without coding. Of interest to me is that we are using correct methods with yielding correct answers. I am not project manager, only responsible for the technical aspects. I look at the program from the point of view of users.

For the next release of the system, we are adding a new "X physics" representation, developed by interviewee US1 and a few more people. From a software point of view, the current code is a throwaway code, used only for validation purposes. It contains old input and output formats and is not productified; after this internal release we will run a strict software project (no new physics) and introduce new input and output.

There are currently two versions of the program, one to be run standalone, and one that is run automatically by another, online system; one additional reason for modifying the system is to make it compatible with this other system.

Structure

(Not a technical expert, ask the other interviewees.)

Framework

Current

We are using Fortran 66 and up. Currently, much is written in F90/95.

Future

All new code is written in Fortran 95.

Components and Conceptual Integrity

Physics Module

Future module

Component. 60 times more data is used in the new model. This is due to the fact that we now use individual data instead of approximations – this is an improved model, closer to reality.

Error Handling and Logging

Future Error Handling and Logging

Reasoning. I know no details but I like the idea of sharing a library across sites and program codes.

Status. In the future version of the system (the software improvement release), the new error handling library (common with Sweden) will be used throughout the program, and this library will also be enforced in all Fortran programs we develop.

Storage

Current Storage

Component. Previous versions has been limited by Fortran 77 where array sizes were statically defined (compiled into the code). There is however a tradeoff between speed and flexibility – with dynamic arrays in Fortran 95 the code is 2-3 times slower. It appears as the newer compiler is not as optimized as the previous, and there are also noticeable differences between target platforms.

Reasoning. There is a clear improvement the way users input data. Much more "object-oriented" input, closer to reality – not so big step to translate from reality to model.

Other Remarks

In the future, several parts will be common between US and Swedish system. I envision that 90% will be identical. The last 10% will be to preserve the individual systems' look-and-feel, plus some fundamental differences between the two domains we are covering.

Interview US3

Background

I have been in the development team since ca 2 years. .

Structure

Current

There are some modules that are specific to the US and the Swedish system, and some that are common.

Framework

(Not really discussed; nothing new compared to interviewee US1.)

Components and Conceptual Integrity

Data Structures

Future Data Structures

Reasoning. To enable sharing the physics modules, the data structures they are dependent on must also be shared. Because these structures change, other modules in the system also need to change. These structures are "good enough" for the future systems. Modifications made are not throw-away, there is little/no wasted effort.

Physics Module

Future module

Reasoning. This module was the starting point for the integration. It needed improvement in both systems, and we wanted to share it.

Error Handling and Logging

Future Error Handling and Logging

Component. Interviewee SE2 developed an error handling system, which we want to use in our system. I think it is written in C++.

Reasoning. Using the new error handling system will cause changes everywhere in all modules. However, it is easy to change all old calls to the new one, almost automatically. There is no use in preserving the old interface and make the old function call the new.

Storage

Future Storage

Component. The input and output to the system has an old interface.

Reasoning. The future system will use something new that meets the common requirements.

Other Remarks

For the future, we need to consider how to make other common physics modules. Either we continue using two different modules, or re-program a new one.

Some 60-70% can be shared, but the last 30% are specific to the two main types of simulations supported.