Investigating the feasibility and worth of migrating legacy systems

Patrick Naish
pn3g10@zepler.net

Abstract—For at least forty years, mainframe computing has been the backbone of technological industry, to a greater or lesser extent. Modernising many of the systems developed on mainframes and other legacy platforms has been a priority for the industry in recent years, as the developers who once maintained those systems have retired, or soon will. This paper explores the options available for updating these legacy systems, and how appropriate and effective these various options are to a company. A comparison is drawn between the various available approaches, to illustrate the breadth of options available and their aptness for particular applications.

1 Introduction

For at least forty years, mainframe computing has been the backbone of technological industry, to a greater or lesser extent. Despite having fallen out of favour in more recent years, in deference to the modern client-server style of computational interaction, mainframe software still underpins a vast amount of modern systems. There were an estimated 200 billion lines of COBOL in use in 2009[1], with no indication of a decline in its use. However, COBOL is widely considered to be a dead language, and the fact that young computer scientists are taught languages such as Java[1] and C++ means that many are unwilling to learn older languages such as COBOL or PL/I. Those developers who do know the languages required to maintain legacy mainframe systems are rapidly reaching the age of retirement and leaving the industry, making the continued use of mainframe systems in their current form impractical, or eventually even impossible.

One of the major challenges in redevelopment is noted by Sneed[2] in that, when replacing an application with which users have grown familiar, they expect the exact same functionality and features (and potentially new features on top of those). Because of this, he says, "...it is much more difficult to replace an existing application than it is to develop one from scratch."[2]. Unfortunately, there is often a lack of resources (monetary and otherwise) available for redevelopment efforts. A 1996 case study[3] estimated that the cost of redeveloping the client's existing systems from scratch would cost approximately $24 million, over four years (with twenty staff), totalling approximately "...100 person years of effort..."[3]. It is therefore evident that there is a need for a means by which to ease the modernisation of legacy systems, without incurring unfeasibly large costs to the maintainers of said systems.

This paper will discuss the current available approaches to modernisation in section 2, the issues which must still be addressed in the field in section 3, and draw a conclusion in section 4.

2 Approaches to modernisation

Almonaies et al.[4] suggest four main approaches towards the modernisation of legacy systems:

- **Replacement**, in which one either rewrites existing applications in a modern language and style, or replaces them altogether (often "...with an off the shelf product..."[4]).
- **Reengineering**, in which reverse-engineering is used to add modern

1. http://fcw.com/articles/2009/07/13/tech-cobol-turns-50.aspx

functionality into the legacy systems themselves.

- **Wrapping**, in which an interface is developed to encapsulate the legacy components so that they may be accessed as services from other components.
- **Migration**, in which the legacy system is transitioned into a new environment "...while preserving the original systemś data and functionality..."[4].

Sneed[5] discusses this in terms of migration versus integration, where migration involves the actual transferral and/or transformation of code and components into a new environment. Integration instead refers to remotely accessing components from the new environment, without transitioning existing components into it.

As previously noted in 1, replacement is potentially either prohibitively expensive[3] or fails due to difficulties users experience in adopting the new software[2]. This is not to say it is never a practical option; for smaller-scale applications, replacement may well be the most cost-effective solution after analysis. For the most part, however, existing literature deals with migration[1]–[3], [5]–[15], with elements of wrapping[10], [15]–[17].

The CelLEST project[18] is discussed in the context of reengineering, although this still makes heavy use of wrapping. Distante et al.[19] also discuss UWA techniques for reengineering for the web.

2.1 Service-Oriented Architectures

The most popular architecture to which legacy systems are migrated is that of a **Service-Oriented Architecture** (SOA)[4], [5], [8], [10], [13], [20]. A service is defined by Perrey and Lycett[21] as "...the boundary between a consumer understandable value proposition and a technical implementation...". That is to say, it provides an interface between an agent and some functionality to allow for interaction without necessitating a deep understanding of underlying code. SOA is, therefore, a design pattern based on the interaction between these services.

Some of the advantages of SOA for the purposes of migrating legacy systems are its flexibility, reusability and abstraction[4]. As Aversano et al. note[6], the mainframe systems were largely built before the internet became prevalent, using a centralised architecture to which a limited number of trusted users would connect. In the modern world, systems are accessed far more widely (e.g. through the internet), by a large number of users, and therefore the architecture must evolve accordingly.

2.2 Tools for code analysis

One of the main tasks in migrating systems is to analyse code for elements which can be extracted and encapsulated, as well as judging the feasibility of migrating a certain piece of code. This would be prohibitively complex (and therefore time-consuming and costly) for a human to perform manually, so tools have been developed to do so automatically (as far as possible).

Sneed[5] uses the **COBAudit** tool to assess a large volume of programs in his pilot project. This took "...56 different size measurements..."[5] from each program, including lines of code and number of data structures. From these, eight metrics were used to determine the programs' overall complexities (as discussed by Sneed in a 1995 paper[22]). These complexity measurements are then normalised using **SoftAudit**, with values above 0.5 indicating high complexity, and those below indicating low complexity. Any values over 0.8 can be considered to indicate extreme complexity. From the normalised results, it is then possible to select programs which are suitable for reuse, as well as identify the sections of them for which this will be most straightforward or difficult.

Lewis et al.[23] use the **Understand for C++**[2] tool to extract both a static analysis of a C++ application, and metrics for assessing the ease of migration. They then use the **Architecture Reconstruction and Mining** (ARMIN)[24] tool to attempt to reconstruct the architecture of analysed programs, in order to find inter-service dependencies (as well as issues with these dependencies in their current form). This

2. http://www.scitools.com/

analysis is then used to create a migration strategy moving forward.

2.3 Tools for migration

Once reusable code has been identified, there are tools available for facilitating migration. One set of these tools is **COB2WEB**, created by Sneed[13], comprising **COBStrip**, **COBWrap**, and **COBLink**, as well as the **COBAudit** tool mentioned in subsection 2.2.

2.3.1 COBStrip

This tool uses code slicing to decompose programs[9]. This concept is discussed in some detail by Weiser[25], and applied practically in the context of migration by Sneed[5], [13]. Its purpose is to separate out various 'slices' from a program, where each slice represents a path through the program that will perform a given function of the original code. These slices are used to "...generate multiple instances of the same program..."[5], [13], which can then be run as individual web services. The tool requires some user interaction, but reduces the workload of slicing significantly.

2.3.2 COBWrap

This tool uses the stripped programs generated by **COBStrip**, and replaces input/output (I/O) operations with "...calls to a wrapper module..."[5], [13], as well as moving required data to the correct sections of the program. This can be performed without any user interaction, as it is a reasonably simple operation when used with predictable programmatic structures (such as those in COBOL).

2.3.3 COBLink

This tool uses the wrapped programs generated by **COBWrap**, and creates **Web Services Description Language** (WSDL)[3] interfaces for both incoming requests and outgoing responses, as well as COBOL modules for translating between the WSDL and COBOL parameters. This process has been described by Sneed in detail[26], as well as applied in a practical application[5], [13]. Once these are generated, it is possible to interact with the services with **Simple Object Access Protocol** (SOAP)[4] messages, thereby accessing the functionality of the legacy application through a modern, flexible interface.

2.4 Techniques and approaches

Due to the maturity of both mainframe systems and the more modern SOA, there has been a lot of time dedicated to exploring techniques for applying modernisation tools and practices to real-world systems. Some of these are purely theoretical, whereas others have been tested in

2.4.1 Service-Oriented Migration and Reuse Technique (SMART)

SMART was described by Lewis et al. as a "...process for evaluating legacy components for their potential to become services..."[23], [27]. As opposed to the tools mentioned in subsection 2.2 and subsection 2.3, this is a set of in-depth analyses of various aspects of the legacy system, target environment, user base and required effort. This is something more of a software engineering approach than a computer science one, as it involves considering the business as a whole and developing strategies, rather than only considering the technical challenge involved. This, of course, leads to a larger amount of work needing to be performed initially, but creates a more cohesive strategy for the overall migration process.

Part of this process may also involve architecture reconstruction, using a tool such as **ARMIN** (previously mentioned in subsection 2.2). This allows for analysis of **Rigi Standard Format** (RSF)[5] files, containing metadata extracted from source files in various languages[24]. These analyses are used to create models and abstractions which, in turn, "...produce higher-levels[*sic*] models and views..."[24]. Eventually, through this process, the "...desired set of views is produced."[24] An evident use of this is to separate out a legacy application into **Model–View–Controller** (MVC) layers, as additionally discussed by Bodhuin et al.[7].

3. http://www.w3.org/TR/wsdl

4. http://www.w3.org/TR/soap

5. http://www.rigi.cs.uvic.ca/downloads/rigi/doc/node52.html

2.4.2 *Service-Oriented Software Reengineering Methodology (SoSR)*

SoSR was proposed by Chung et al. as an "...architecture-centric, service-oriented, role-specific and model-driven"[28] methodology for software reengineering. It uses a '4+1 view model', where a system is considered to be composed of four primary views:

- **Design view**
- **Implementation view**
- **Process view**
- **Deployment view**

which share a single **use case view**. The use case view represents the software engineering aspects of the system architecture, such as user requirements; the four primary views represent the logical and physical, static and dynamic aspects of the overall system. It also makes use of **Responsible, Accountable, Consulting and Informed** (RACI) charts[6] to show the tasks required for each role in the system.

2.4.3 *Ubiquitous Web Applications Design Framework (UWA)*

UWA[29] and the **Extended UWA Transaction Design Model** (UWAT+)[30] are discussed by Distante et al.[19] as a means to reengineer legacy systems. As with **SoSR** (subsubsection 2.4.2) and **SMART** (subsubsection 2.4.1), this deals not only with the technical aspect of system modernisation, but also the software engineering aspects such as requirement gathering. UWAT+ has the concept of "...Web transaction[s]..."[19], which are a set of user interactions required to complete a given task.

This relies fairly heavily on reverse-engineering to generate 'meta-models', which can then be used to generate more concrete models in turn. From these, a web version of the original system can be engineered.

2.4.4 *CelLEST*

The CelLEST project[7] deals mostly with front-end interfaces, and user interactions with them, for a given system. Stroulia et al.[18] discuss that, as opposed to techniques based on gaining an understanding of the original application's architecture through code analysis, CelLEST constructs a model of the behaviour of the legacy system through analysing user-interface interaction paths. This model is then used as an interface between users and the legacy application (i.e. wrapping), without modifying the original code.

Because of the nature of the analysis, this requires a large amount of user interaction, ideally by "an expert user of the legacy application"[18]. However, it does not require the full software-engineering processes that are used by techniques such as **SMART** (subsubsection 2.4.1). Stroulia et al. note that this technique cannot be used for maintenance, as it does not modify the underlying code, but is a means of providing access to said code through a modern interface.

6. http://racichart.org/
7. http://webdocs.cs.ualberta.ca/~stroulia/CELLEST/new/

2.4.5 *Finite State Automata*

Canfora et al.[8], [10] discuss the usage of a finite state automata to model the interactions between a user and the legacy system, on the basis that a traditional purely 'black-box' approach to wrapping will not be aware of the current state of the legacy system when executing a command, which may lead to unknown behaviour (a problem exacerbated by the scaling of the number of users simultaneously accessing a system, from the mainframe to SOA). Thus, the automaton for a given system will keep track of the current state of the underlying system, and model the currently available I/O operations through a wrapper layer. Paths through the automaton can be identified in a similar manner to code slicing (subsubsection 2.3.1), and used to assess whether all the scenarios for a given operation have been accounted for.

2.5 Comparison of approaches

Find best approach to comparison, discuss case studies

3 ISSUES WHICH MUST BE ADDRESSED

Even with the currently available tools and techniques, the problem of modernising legacy systems is far from solved. The main issues which present themselves are those discussed below:

- **Cost-effectiveness**
- **Maintainability**
- **Risk**
- **Automation**
- **Testing**

as well as more human issues such as user-acceptance and the necessity for retraining. It is also important to consider that, in a real-world business scenario, deployment of new systems must be approached gradually in order to not disrupt current operation[3], [4], [13]. This itself adds another dimension of complexity to the entire effort of modernisation.

3.1 Cost-effectiveness

The best example of cost-effective migration was discussed in section 1, with the case study from Duncan et al.[3] showing an estimated $24 million cost for redevelopment, as opposed to a $4.8 million cost for migration (including hardware, software, training and operating costs) over three years. Additionally, there was a benefit to the company of $9.45 million over that same three year period, due to the improvements made during modernisation.

Although this was a success, the lower cost was due to the overall complexity of the systems that required modernisation. With a less complex legacy system, the effort required to redevelop may be significantly less than the effort required to migrate or reengineer. This falls, therefore, to analysts to determine the most appropriate course of action – but even this brings its own expenses and consumes resources.

3.2 Maintainability

While some of the approaches do not even attempt to deal with the issue of maintainability[18], most make some attempt to ease further development. This is achieved by updating the original code and wrapping it in a modern language, transforming the original code into a more modern language, or some combination of the two. The rationale is that younger computer scientists and software engineers will be more familiar with languages such as Java[1], and therefore will be able to continue maintaining the software moving forwards.

This does not deal with the fact that design patterns have changed significantly even over the last twenty years, let alone forty, and modernisation is something of a stopgap solution to updating systems. It could be considered that the entire exercise is striking a balance between cost and maintainability, in that a larger expenditure will most likely leave you with a more modern and maintainable solution. Obviously this is not always feasible or even appropriate for a smaller business, but it is important to remember that no system can be maintained indefinitely while competing successfully against more modern alternatives. It may also be the case that redeveloped software will better suit an organisation's needs.

Maintaining a system without a deep understanding of it also presents some issues. Lack of familiarity introduces more complexity into maintaining it, especially when one is now required to not only understand the modern and legacy code (where it remains), but also the interactions between them. However, a similar argument for lack of familiarity could be made against entirely replacing an existing system[4].

3.3 Automation

The tools discussed in subsection 2.3 and subsection 2.2 are a step towards automating the modernisation process, thereby reducing the costs required to do so. However, due to the varied and complex nature of legacy systems, it is extremely difficult to fully automate many steps. Sneed has made many steps towards automating the process as far as possible for certain languages[1], [2], [5], [13], [15], [26], as have other researchers[6], [14], [16], [19], [24], [31], but the problem is far from solved. It may be the case that this is an issue which may never be fully overcome, as there is an almost infinite number of potential edge cases for any software, but it does at least pave the way for automation as far as possible.

4 Conclusion

Conclusion

Acknowledgements

The author would like to thank Prof. Michael Butler for his supervision during the writing of this paper.

References

[1] H. M. Sneed and K. Erdoes, "Migrating AS400-COBOL to Java: A Report from the Field," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 231–240. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6498471

[2] H. Sneed, "Migrating from COBOL to Java," *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–7, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5609583

[3] D. G. Duncan and S. B. Lele, "Converting from mainframe to Client/Server at Telogy Inc," *Journal of Software Maintenance-Research and Practice*, vol. 8, no. 5, pp. 321–344, 1996. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1096-908X(199609)8:5⟨321::AID-SMR135⟩3.0.CO;2-4

[4] A. Almonaies, J. Cordy, and T. Dean, "Legacy system evolution towards service-oriented architecture," in *International Workshop on SOA Migration and Evolution (SOAME 2010)*, 2010, pp. 53–62. [Online]. Available: http://research.cs.queensu.ca/home/cordy/Papers/ACD_MigToSOA_SOAME10.pdf

[5] H. M. Sneed, "A pilot project for migrating COBOL code to web services," pp. 441–451, 2009.

[6] L. Aversano, G. Canfora, A. Cimitile, and A. D. Lucia, "Migrating legacy systems to the Web: an experience report," *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[7] T. Bodhuin, E. Guardabascio, and M. Tortorella, "Migrating COBOL systems to the Web by using the MVC design pattern," *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002.

[8] G. Canfora, A. Fasolino, G. Frattolillo, and P. Tramontana, "Migrating interactive legacy systems to Web services," *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006.

[9] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, "Decomposing legacy programs: A first step towards migrating to client-server platforms," *Journal of Systems and Software*, vol. 54, pp. 99–110, 2000.

[10] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, "A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures," *Journal of Systems and Software*, vol. 81, pp. 463–480, 2008.

[11] A. D. Lucia, G. D. Lucca, A. Fasolino, P. Guerra, and S. Petruzzelli, "Migrating legacy systems towards object-oriented platforms," *Proceedings International Conference on Software Maintenance*, 1997.

[12] G. Lewis, E. Morris, and D. Smith, "Analyzing the reuse potential of migrating legacy components to a service-oriented architecture," *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006.

[13] H. Sneed, "COB2WEB a toolset for migrating to web services," *2008 10th International Symposium on Web Site Evolution*, 2008.

[14] L. W. L. Wu, H. Sahraoui, and P. Valtchev, "Coping with legacy system migration complexity," *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, 2005.

[15] H. Sneed, "Encapsulating legacy software for use in client/server systems," *Reverse Engineering, 1996., Proceedings of the . . .*, pp. 104–119, 1996. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=558885

[16] C.-C. Chiang, "Wrapping legacy systems for use in heterogeneous computing environments," pp. 497–507, 2001.

[17] H. Sneed and S. Sneed, "Creating Web Services from Legacy Host Programs," in *Web Site Evolution, 2003. Theme: Architecture. Proceedings. Fifth IEEE International Workshop on*, 2003, pp. 59–65.

[18] E. Stroulia, M. El-Ramly, and P. Sorenson, "From legacy to Web through interaction modeling," *Software Maintenance, 2002. Proceedings. International Conference on*, pp. 320–329, 2002.

[19] D. Distante, S. Tilley, and G. Canfora, "Towards a holistic approach to redesigning legacy applications for the Web with UWAT+," *2011 15th European Conference on Software Maintenance and Reengineering*, vol. 0, pp. 295–299, 2006.

[20] A. Koschel, C. Kleiner, and I. Astrova, "Mainframe Application Modernization Based on Service-Oriented Architecture," *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 298–301, Nov. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5359601

[21] R. Perrey and M. Lycett, "Service-oriented architecture," *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pp. 116–119, 2003. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1210138

[22] H. Sneed, "Understanding software through numbers: A metric based approach to program comprehension," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 405–419, 1995. [Online]. Available: http://www3.interscience.wiley.com/journal/113446591/abstract

[23] G. Lewis, E. Morris, L. O'Brien, D. Smith, and L. Wrage, "SMART: The service-oriented migration and reuse technique," Tech. Rep. September, 2005. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA441900

[24] L. O'Brien, D. Smith, and G. Lewis, "Supporting Migration to Services using Software Architecture Reconstruction," *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pp. 81–91, 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1691635

[25] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, 1984.

[26] H. Sneed, "Wrapping legacy COBOL programs behind an XML-interface," *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pp. 189–197, 2001.

[27] G. Lewis, E. Morris, and D. Smith, "Service-Oriented Migration and Reuse Technique (SMART)," *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pp. 222–229, 2005.

[28] S. C. S. Chung, J. B. C. A. J. B. C. An, and S. D. S. Davalos, "Service-Oriented Software Reengineering: SoSR," *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pp. 172c–172c, 2007.

[29] UWA Consortium, "Ubiquitous Web Applications," *Proceedings of the eBusiness and eWork Conference 2002*,

2002. [Online]. Available: http://home.deib.polimi.it/baresi/papers/eWork02.pdf

[30] D. Distante and S. Tilley, "Conceptual modeling of web application transactions: towards a revised and extended version of the UWA transaction design model," *Multimedia Modelling Conference, 2005. MMM 2005. Proceedings of the 11th International*, pp. 439–445, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1386027

[31] A. V. Deursen and L. Moonen, "Type inference for COBOL systems," *Reverse Engineering, 1998. . . .*, 1998. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=723192