Software Systems Integration and Architectural Analysis – A Case Study

Rikard Land, Ivica Crnkovic

Mälardalen University
Department of Computer Science and Engineering
PO Box 883, SE-721 23 Västerås, Sweden
+46 21 10 70 35, +46 21 10 31 83

{rikard.land, ivica.crnkovic}@mdh.se
http://www.idt.mdh.se/{~rld, ~icc}

Abstract

Software systems no longer evolve as separate entities but are also integrated with each other. The purpose of integrating software systems can be to increase user-value or to decrease maintenance costs. Different approaches, one of which is software architectural analysis, can be used in the process of integration planning and design.

This paper presents a case study in which three software systems were to be integrated. We show how architectural reasoning was used to design and compare integration alternatives. In particular, four different levels of the integration were discussed (interoperation, a so-called Enterprise Application Integration, an integration based on a common data model, and a full integration). We also show how cost, time to delivery and maintainability of the integrated solution were estimated.

On the basis of the case study, we analyze the advantages and limits of the architectural approach as such and conclude by outlining directions for future research: how to incorporate analysis of cost, time to delivery, and risk in architectural analysis, and how to make architectural analysis more suitable for comparing many aspects of many alternatives during development. Finally we outline the limitations of architectural analysis.

Keywords
Architectural Analysis, Enterprise Application Integration, Information Systems, Legacy Systems, Software Architecture, Software Integration.

1. Introduction

The evolution, migration and integration of existing software (legacy) systems are widespread and a formidable challenge to today's businesses [4,19]. This paper will focus on the *integration* of software systems. Systems need to be integrated for many reasons. In an organization, processes are usually supported by several tools and there is a need for integration of these tools to achieve an integrated and seamless process. Company mergers demand increased interoperability and integration of tools. Such tools can be very diverse with respect to technologies, structures and use and their integration can therefore be very complex, tedious, and time- and effort-consuming. One important question which arises: Is it feasible to integrate these tools and which approach is the best to analyze, design and implement the integration?

Architecture-centered software development is a well-established strategy [2,3,13,21]. We have experienced the architecture of a system as an appropriate starting point around which to concentrate integration activities. One common experience is that integration is more complex and costly than first expected due to "architectural mismatches" [10,11], and this problem should be addressed at the architectural level. It also seems possible that some architectural analysis techniques used during new development could also be applicable during system evolution and integration. In this paper we show the extent to which an architecture-centric approach can be used during system evolution and integration, and how accurate and relevant the result of such an architecture-based analysis is.

Our aim has been to present our experiences from a case study in which three software systems were to be integrated after a company merger. We have monitored the decision process, and the actual integration has just begun. The activities were focused around the systems' architectures. We describe the three integration approaches that were discerned and discussed, how architectural descriptions of the two most interesting were developed and analyzed and the decisions taken for the development project. Further we analyze the proposed solutions showing the strong and weak sides of the architectural strategy as such.

The rest of this paper is organized as follows. Section 2 provides the background of our case study, section 3 discusses four integration approaches, and section 4 uses

the case study to elaborate on architectural analyses possible during system integration. Section 5 describes related work, and section 6 concludes the paper and suggests directions for future research.

2. Introducing the Case Study

The case study concerns a large North American industrial enterprise with thousands of employees that acquired a smaller (approximately 800 employees) European company operating in the same business area. Software, mainly developed in-house, is used for simulations and management of simulation data, i.e. as tools for development and production of other products. The functionality of the software developed in the two organizations prior to the merger was found to overlap to some extent, and three systems suitable for integration were identified. A project was launched with the aim of arriving at a decision on strategic principles for the integration, based on the proposed architecture for the integrated system. This was the first major collaboration between the two previously separate software departments.

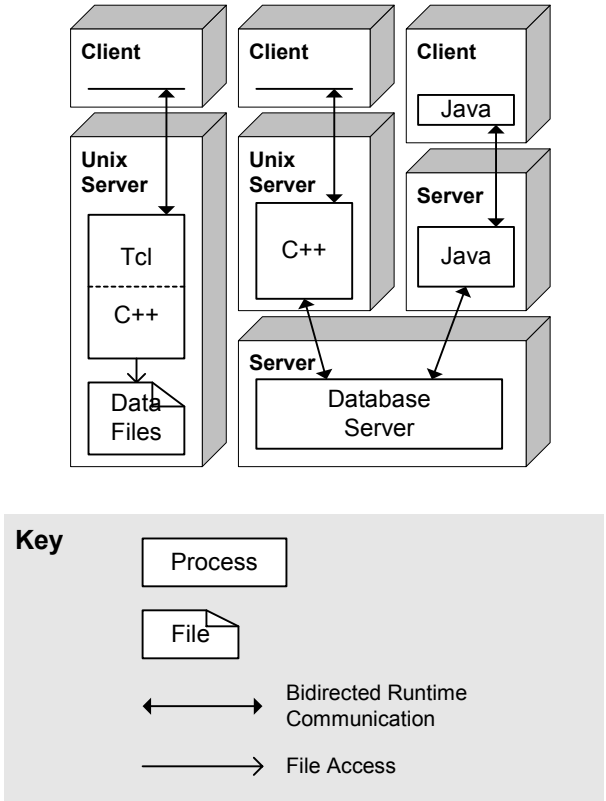

Figure 1. Today's three systems.

Figure 1 describes the existing systems' architectures in a simplified manner in a high-level diagram combining an execution view of the system with the code view [2,7,13,16]. The sizes of the rectangles indicate the relative sizes of the components of the systems (as measured in lines of code). One system uses a proprietary object-oriented database, implemented as files accessed through library functions, while the other two systems, which were developed at the same site, share data in a common commercial relational database executing as a database server. The most modern system is built with three-tier architecture in Java 2 Enterprise Edition (J2EE), while the two older systems are developed to run in a Unix environment with only a thin X Windows client displaying the user interface (the "thin" client is denoted by a rectangle with zero height in the figure). These are written mostly in Tcl and C++, and C++ with the use of Motif. The "Tcl/C++ system" contains ~350 KLOC (thousands of lines of code), the "C++/Motif system" 140 KLOC, and the "Java system" 90 KLOC.

3. Integration Approaches

When developing architectures of new systems, the main goal is to achieve the functionality and quality properties of the system in accordance with the specified requirements and identified constraints. When, however, existing systems are to be integrated, there may be many more constraints to be considered: backward compatibility requirements, existing procedures in the organization, possible incompatibility between the systems, partial overlap of functionality, etc. Similarly, the integrated system is basically required to provide the same functionality as the separate systems did previously, but also, for example, to ensure data consistency and enable automation of certain tasks previously performed manually. When developing new software, it is possible to design a system that is conceptually integrated [5] (i.e. conforms to a coherent set of design ideas), but this is typically not possible when integrating software since the existing software may have been built with different design concepts [11]. Another problem is how to deal with the existing systems during the integration phase (and even long after, if they have been delivered and are subject to long-term commitments). This problem becomes more complex the more calendar-time the integration will take as there is a pronounced tradeoff between costs in the short term and in the long term when different integration solutions have different maintainability characteristics. For example, there is an opportunity to replace older with more recent technologies to secure the system usability for the future. Scenarios possible if the systems are not integrated should also be considered.

In the analysis and decision process we have discerned four integration approaches or "levels" with different characteristics. They are:

- **Interoperability through import and export facilities.** The simplest form of using services between tools is to obtain interoperability by importing/exporting data and providing services. The

data could either be transferred manually when data is needed, or automatically. To some extent, this could be done without modifying existing systems (e.g. if there is a known API or it is possible to access data directly from the data sources), and if source code is available it is possible to add these types of facilities. This approach would allow information to flow between the systems, which would give users a limited amount of increased value. It would be difficult to achieve an integrated and seamless process, as some data could be generated by a particular tool not necessarily capable of automatic execution. Moreover, there would be problems of data inconsistency.

- **Enterprise Application Integration (EAI).** Many systems used inside a company are acquired rather than built, and it is not an option to modify them. Such systems are used within a company, as opposed to the software products a company not only uses but also manufactures and installs at customers' sites. Integrating such enterprise software systems involve using and building wrappers, adapters, or other types of connectors. In such a resulting "loose" integration the system components operate independently of each other and may store data in their own repository. Depending on the situation, EAI can be based on component technologies such as COM or CORBA, while in other cases EAI is enabled through import and export interfaces (as described in previous bullet). Well-specified interfaces and intercommunication services (middleware) often play a crucial role in this type of integration.
- **Integration on data level.** By sharing data e.g. through the use of a common database, the users will benefit from access to more information. Since the systems store complementary information about the same data items; the information will be consistent, coherent and correct. However, it would presumably require more effort to reach there: a common data model must be defined and implemented and the existing systems must be modified to use this database. If this is done carefully, maintenance costs could be decreased since there is only one database to be maintained and there are opportunities to coordinate certain maintenance tasks. On the other hand, maintenance becomes more complex since the database must be compatible with three systems (which are possibly released in new versions independently). Also data integration may have an impact on code change, due to possible data inconsistencies or duplicated information.
- **Integration on source code level.** By "merging" source code, the users would experience one homogeneous system in which similar tasks are performed in the same way and there would be only one database (the commercial database used today by the C++/Motif system and the Java system). Future maintenance costs can be decreased since it would be conceptually integrated, and presumably the total number of lines of code, programming languages, third-party software and technologies used will decrease. Most probably the code integration would require integration of data.

Interoperability through import and export facilities is the most common way of beginning an integration initiative [8]. It is the fastest way to achieve (a limited amount of) increased functionality and it includes the lowest risk of all alternatives, which is the reason why managers usually adopt this approach. In a combination with a loose integration (EAI) it can provide a flexible and smooth integration process of transition: the import/export facilities can be successively replaced by communicating components and more and more integrated repositories. Of course, this approach has its disadvantages – in total it will arguably require more effort, and the final solution may technically not be as optimized as the results of the "data level" or "code level" approaches. This of course depends on the goals of the integration.

Which integration approach to use in a particular context depends not only the objective of the integration, but also e.g. the organizational context and whether source code is available or not. For example, is the goal to produce an integrated product for the market, or is the system to be used only in-house? Is integration of software a result of a company merger? Is integration expected to decrease maintenance costs or to increase the value for users (or both)? Who owns the source code? Can the systems to be integrated be expected to be released in subsequent versions by (other) independent vendors? Is modifying source code an option, considering both its availability and possible legal restrictions? Business constraints also limit the possibilities – the resources are limited and time to market an important concern. One must also consider the risks associated with each alternative, meaning the probability of overrunning budget and/or schedule or not succeed with the integration. The risk parameters include not only those related to technical problems, but also those associated with the collaboration of two software development departments which had previously belonged to different companies and only recently began collaborating.

The project team of the case study intuitively felt that the benefits and the cost of implementation, the time to delivery, and the risk of the integration approaches described above should be related roughly as shown in Figure 2. The diagram is very simplistic assuming there is only one "benefit" dimension, but as mentioned earlier there may be different types of goals for integration, such as increased usability or decreased maintenance costs. EAI

was never explicitly considered as a separate approach during the case study and is therefore omitted from the figure.

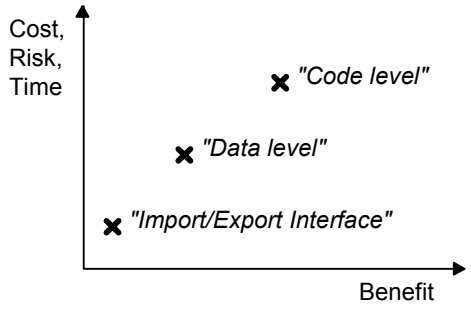

Figure 2: Expected relations between risk, cost, and time to delivery.

4. Development of Integration Alternatives

Developers from the two sites met and analyzed the existing systems at the architectural level, and then developed and analyzed two integration alternatives. The developers had architected, implemented and/or maintained the existing systems and were thus very experienced in the design rationale of the systems and the technologies used therein. The architectural alternatives were then handed over to management to decide which alternative should be used. The integration process was based on IEEE standard 1471-2000 [14] and is described in more detail in [17,18].

The "import/export level" interoperability was not discussed in any depth since it was apparent that more benefits were desired than could be expected with this approach. Instead, the software developers/architects tried the other approaches to integration, by conceptually combining the source code components of the existing system in different ways. The existing documentation had first to be improved by e.g. using the same notation (UML) and the same sets of architectural views (a code view and an execution view were considered sufficient) to make them easy to merge [18]. Each diagram contained about ten components, sufficient to permit the kind of reasoning that will be described. By annotating the existing components with associated effort, number of lines of code, language, technologies, and third-party software used, the developers could reason about how well the components would fit together. During the development of alternatives, statements about the quality properties of the integrated system such as performance and scalability were based on the characteristics of the existing systems. Patterns known to have caused deficiencies and strengths in the existing systems in these respects made it possible to evaluate and discard working alternatives rapidly. The developers had a list of such concerns, to ensure that all those of importance were addressed. The process of developing and refining alternatives and analyzing them was more iterative than is reflected in the present paper where we only present two remaining alternatives and the analyses of three specific concerns in more detail (sections 0 through 4.3).

The two remaining main alternatives conformed well to the "data level" and the "code level" integration approaches. Both these alternatives would necessarily need a common data model and shared data storage. From there, the two different levels of integration would require different types of actions: for "data level" integration, the existing systems would need to be modified due to changes in the data model, and for "code level" integration, much of the existing functionality would need to be rewritten in Java; see Figure 3. In reality, these descriptions were more detailed than the figure suggests; About ten components were used in each of the same two views for describing the existing systems, a code view and an execution view.

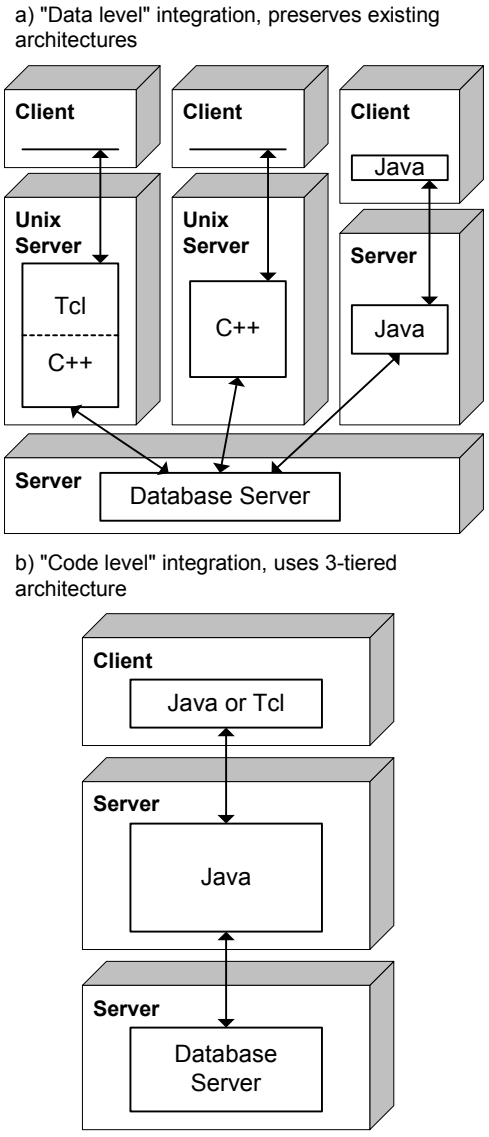

Figure 3. The two main integration alternatives.

Architectural descriptions such as these make it possible to reason about several properties of the resulting integrated system.

4.1 Future Maintainability

The following factors were considered in the case study to be able to compare the future maintenance costs of the integration alternatives:

- **Technologies used.** The number of technologies used in the integrated system arguably tells something about its complexity. By technologies we more specifically mean the following: programming languages, development tools (such as code generators and environments), third-party software packages used in runtime, and interaction protocols. Too many such technologies will presumably create maintenance difficulties since maintaining staff needs to master a large number of languages and specific products and technologies, but at the same time tools and third-party software should of course be used whenever possible to increase efficiency. A reasonable number must therefore be estimated in any specific case. In our case study, the total number of languages and technologies used in the "code level" alternative would be reduced to 6 to 8 languages instead of the 11 found in the existing system combined, a number which would be preserved in the "data level" alternative. The number of third-party packages providing approximately the same functionality could be reduced from 9 to 5, and two other technologies would also become superfluous.
- **LOC.** The total number of lines of code (LOC) has been suggested as a measure of maintainability; it is e.g. part of the Maintainability Index (MI) [20,23]. In the case study, the total number of lines of code would be considerably less with the "code level" alternative. No numbers were estimated, but while the "code level" alternative would mean that code was merged and the number of lines of code would be less than today, the "data level" alternative would rather raise the need of duplicating more functionality in the long term.
- **Conceptual integrity.** Although a system commonly implements several architectural styles at the same time – "heterogeneous systems" [2] – this should come as a result of a conscious decision rather than fortuitously for the architecture to be conceptually integrated [5]. In the case study, it was clear, by considering the overall architectural styles of the systems, that the "data level" alternative involved three styles in parallel while the "code level" would reflect a single set of design ideas.

It might seem surprising that in the case study, in the "code level" integration alternative, the server is written totally in Java. Would it not be possible to pursue the EAI approach and produce a loosely integrated solution, involving the reuse of existing parts written e.g. in C++? With the platform already in use, J2EE, it would be possible to write wrappers that "componentized" different parts of the legacy code. This was considered, and, by iteration the architectural description of this alternative was modified and analyzed with respect to the cost of implementation. Based on these estimates, all solutions involving wrappers and componentization were ultimately discarded and only the two alternatives already presented remained.

Whether to use Java or Tcl in the client for the "code level" alternative was the subject of discussion. Much more user interface code was available in the Tcl/C++ system than in the Java system which was preferable for other reasons. The pros and cons of each alternative were hard to quantify, and eventually this became a question of cost, left to the management to decide.

4.2 Cost Estimation

Estimating the cost of implementing an integrated system based on an architectural description is fairly straightforward. Based on previous experience, developers could estimate the effort associated with each component, considering whether it will remain unmodified, be modified, rewritten, or totally new in the integrated system. Clearly, the outcome of this type of estimation is no better than the estimations for individual components. The advantage of estimation at the component level is that it is easier to grasp, understand, and (we argue) estimate costs for smaller units than for the system as a whole.

This estimation is fairly informal and mainly based on experience, but it can be considered reasonable. First, the developers in the case study were very experienced in the existing systems and software development, second, the developers themselves agreed on the numbers, third, these numbers were higher than the management had expected (implying it not being overly optimistic/unrealistic), fourth, management explicitly asked the developers during the development of the alternatives to find cheaper (and faster) alternatives, something they were unable to do – the only alternative according to them would be the import and export facilities (for the interoperability approach). When summing the effort associated with all components in each alternative the developers found (partly to their surprise) that the implementation costs would be the same for both alternatives (the total estimated times differed by only 5%, which is negligible for such early, relatively rough estimations). This was true for the variant of the "code level" alternative if Tcl was chosen for the client part - using Java would require more resources. The apparently high cost of the "data level" alternative was due to the definition of a common data model, and in the case of the

Tcl/C++ system the use of a new database (a commercial relational database instead of an object-oriented proprietary database). These changes would ripple through the data access layer, the classes modeling the items in the database, and to a limited extent the user interface. Since the total number of lines of code is much greater than the estimated number of lines of code in the "code level" integration alternative, the apparently lower cost of modifying code instead of rewriting it would be nullified by the larger number of lines of code. It would also be necessary to write some new components in two languages.

Bridging solutions would be required and functionality duplicated in both C++ and Java by the existing code (and added to by the development of new functionality and the modifications of e.g. data access layers). When the developers estimated the costs associated with using both Tcl and Java in the client (since much code could be reused), and using only one (thus extending the existing code in one language with the functionality of the other), it was concluded that using two different languages in the client would probably be more costly than using either one, due to the same arguments as above. Some generic components, among them non-trivial graphical widgets, would need to be written in two languages.

Building a common data model from existing data models is one of the major challenges of software engineering [1,10], which was apparent from the cost estimations. We cannot claim, on the basis of a single case study, that the "data level" approach will always be as expensive as the "code level" approach, but this reasoning gives at hand that in general, neither approach is cheap, once a minimum of data level integration is decided upon. For the "data level" alternative this requires changes throughout the existing systems and the "code level" alternative requires changes, to adapt to both the new data model and a single set of technologies, languages, and architectural styles.

4.3 Estimated Time to Delivery

The resulting project plans developed in the case study are shown in Figure 4. Although the diagrams presented here are somewhat simplified compared with those developed in the project, they suffice to illustrate some features of this type of project plan:

- The definition of a common data model is crucial in both integration approaches, since most other activities are dependent on it. In the case study, the developers were explicit that this activity should not be rushed, and should involve the most experienced users as well as developers.

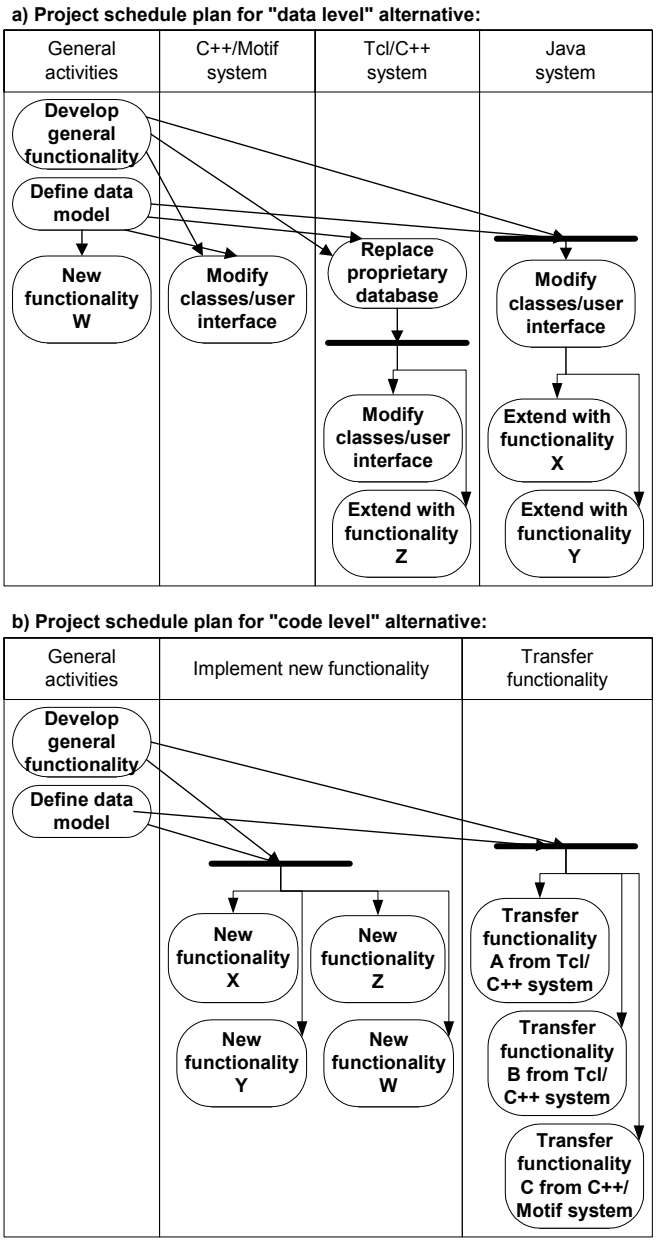

Figure 4: The outlined project plans.

- Management is given a certain amount of freedom by not assigning strict dates to activities. Activities can be prioritized and reordered, and deliveries "spawned off" to meet business demands. More staff can be assigned to certain activities to increase parallelism and throughput. Based on which components would need to be included in a delivery, it is possible to define activities that produce these components; for example, if a delivery with functionality "X" is desired, the activity "Extend with functionality X" or "New functionality X" (for the two alternatives respectively) must be performed as well as all activities on which it

is dependent. One strategy could be to aim at delivering a "vertical slice" of the system, incorporating the functionality that is most used first. In this way some users can begin using the new system, thus minimizing the need for maintenance and development of the existing systems (which will soon be retired).

- In the "code level" alternative, many activities are of the "transfer functionality" type. In this way, users of the Java system will only see the functionality grow rapidly, but the users of the other systems will experience a period when most of the functionality exists in both the system with which they are familiar and the new system. For the "data level" alternative, the activities are more of the kind "modify the existing systems". The users would then continue using their familiar system but, when beginning to use the other systems, would have access to more functionality working on the same data. This type of reasoning impacts on long-term planning aspects such as the time at which existing systems can be phased out and retired.
- In the "code level" alternative, it was possible to identify more general components that would require an initial extra amount of effort and calendar-time but would eventually make the project cheaper and faster. In the "data level" alternative, only few such components were identified.
- Some development of totally new functionality demanded by users was already planned and could not be delayed until the systems integration efforts were completed. However, it was agreed that these activities should be delayed as long as possible – at least until one of the integration alternatives was chosen, and if possible, until the new data model had been defined, and even general components implemented in the case of the "code level" alternative. This was to avoid producing even more source code that would need to be modified during the integration.

4.4 The Decision

When the developers from the two sites had jointly produced these two alternatives and analyzed them, the management was to decide which alternative to choose. It was agreed that the "code level" alternative was considered to be superior to the "data level" alternative from virtually all points of view. The users would experience a more powerful, uniform and homogeneous system. It would also be easier (meaning less costly) to maintain. The analysis had shown that it would include a smaller code base as well as a smaller number of languages, third-party software, and other technologies. The languages and technologies used were more modern, implying that they would be supported by more tools, easier to use and more attractive to potential employees. Not least, the resulting product would be conceptually integrated. Regarding the choice between using Java and Tcl in the client, the management accepted that if the "code level" was decided upon, Tcl would be used since using Tcl implied a significantly smaller effort (due to a larger code base to reuse).

When management considered all this information, they judged the integration to be sufficiently beneficial to motivate the high cost. The benefits included, as we have indicated earlier, increased user efficiency, decreased maintenance costs (in the case of the "code level" alternative), as well as less tangible business advantages such as having an integrated system to offer customers. Also, the evolution scenarios for the existing systems if no integration was performed would be costly; for example, the European organization would probably replace in the near future, the proprietary object-oriented database with a commercial relational database for maintenance and performance reasons. The cost of implementing the "data level" and "code level" alternatives (when using Tcl in the client) had been estimated to differ insignificantly, and as the organization had to develop it with a limited number of staff, the estimated time to delivery would also be very similar, although the deliveries would be of different kinds due to the different natures of the activities needed for the two alternatives. The relation benefit vs. cost and time to delivery can therefore be visualized as Figure 5 illustrates (the "import/export interface" level was not analyzed, hence the parentheses).

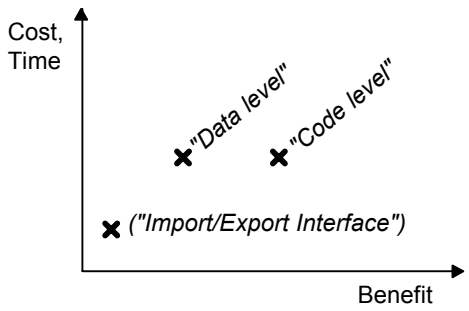

Figure 5: The estimated cost and time to delivery.

As became clear by now, it was less important to get as much benefit as possible for the cost than to decrease the risk as much as possible. No formal risk analysis was performed at this point, but the risk was judged to be higher for the "code level" alternative, since it involves rewriting code that already exists and works, i.e. risking overrunning schedule and budget and/or decreasing the quality of the product, but also a risk in terms of "commitment required" from the departments of two previously separate organizations, not yet close collaborators. By choosing the "data level" alternative, each system would still be functioning and include more

functionality than before, should the integration be discontinued due to e.g. an unacceptable schedule and/or budget situation. This is discernible in the project plans of Figure 4. Management doubted that the cost of the two alternatives would really be similar; they intuitively assumed that the higher benefit, the more effort was required (cost and time), as was sketched in Figure 2. Still, they were explicit in that the *risk* was the decisive factor and not *cost*, when choosing the "data level" alternative.

5. Related Work

There are suggestions that project management during ordinary software development has much to gain from being "architecture-centric" [21]. We have shown some ways of pursuing the architecture-centric approach during integration also. The rest of this section will focus on two related aspects of this, the literature relating to integration approaches, and methods and analysis techniques based on architectural descriptions.

Of the four integration approaches we have discussed, Enterprise Application Integration (EAI) seems to be the most documented [9,12,15,19,22]. This approach concerns in-house integration of the systems an enterprise *uses* rather than *produces*. Johnson [15] uses an architectural approach to analyze the integration of enterprise software systems. In spite of the difficulty of accurately describing the architecture of this type of system because the available documentation is inadequate, architectural analysis can be successfully applied to the design of enterprise systems integration. Johnson has also examined the limitations of architectural descriptions which one must be aware of, limitations that were also experienced in the case study.

None of the architectural methodologies available were completely feasible for the task. The *Architecture Trade-off Analysis Method* (ATAM) [6] and the *Software Architecture Analysis Method* (SAAM) [2,6] are based on stakeholder-generated scenarios. The ATAM requires business drivers and quality attributes to be specified in advance and more detailed architectural descriptions to be available. In the case study, all of this was done in a more iterative manner. Also, with limited resources, it would be impossible to evaluate and compare several alternatives, it being too time-consuming to investigate all combinations of quality attributes for all working alternatives. While both SAAM and ATAM use scenarios to evaluate maintainability, we used another, if less accurate measurement method, comparing the number of lines of code, third-party software, languages, and technologies used, assuming that the lower the number, the easier the maintenance . The *Active Reviews for Intermediate Designs* method (ARID) [6] builds on Active Design Reviews (ADR) and incorporates the idea of scenarios from SAAM and ATAM. It is intended for evaluating partial architectural descriptions, exactly that which was available during the project work. However, it is intended as a type of formal review involving more stakeholders and this was not possible because the project schedule was already fixed, and too tight for an ARID exercise. All of these methodologies analyze functionality (which was relatively trivial in the case study as the integrated system would have the functionality of the three systems combined) and quality attributes such as performance and security (which are of course important for the product of the case study, but considered to be similar to the existing systems) – but none addresses cost, time to delivery, or risk, which were considered more important. The project therefore relied more on the analysts' experience and intuition in analyzing functionality and quality attributes (because of the project's limited resources), and cost, time to delivery, and risk (because there are no available lightweight methodologies for analyzing these properties from architecture sketches).

6. Conclusions

We have shown the central role of software architecture in a case study concerning the integration of three software systems after a company merger. Some important lessons we learned from this case study can be formulated as follows:

- There are at least four approaches available to a software integrator: Enterprise Application Integration (EAI), interoperability, data level integration, and source code integration. The choice between these is typically based on business or organizational considerations rather than technical.
- When the architectural descriptions of existing systems are not easily comparable, the first task is to construct similar architectural descriptions of these. The components of the existing systems can then be rearranged in different ways to form different alternatives. The working alternatives can be briefly analyzed, largely on the basis of known properties of architectural patterns of the existing systems.
- The functional requirements of an integrated system are typically a combination of the functionality of the existing systems, and are relatively easy to assess as compared with other quality attributes.
- The effort required to implement each component of the new system can be estimated in terms of how much can be reused from the existing systems and how much must be rewritten. The total cost of the system is easily calculated from these figures.
- According to the estimations performed in the case study, source code level integration is not necessarily more expensive than data level integration.
- Architectural analysis, as it was carried out in the project, fails to capture all business aspects important

for decisions. All the information needed to produce a project schedule is not present in an architectural description. The *risk* associated with the alternatives was identified as the most important and least analyzed decision criteria.

There are a number of concerns that must be addressed during integration planning as well as during software activities in general. These include the process and time perspective (e.g. will the integration be carried out incrementally, enabling stepwise delivery and retirement of the existing systems?), the organizational issues (e.g. who are the stakeholders?), the cost and effort requirements (e.g. are only minimal additional efforts allowed?), etc. We have shown how a system's architecture can be used as a starting and central point for a systematic analysis of several features. To what extent can such concerns be addressed by architectural analysis? Perhaps the focus on the architecture, basically a technical artifact poses a risk to these other concerns? We have presented means of estimating cost and time of implementation based on architectural descriptions, including outlining project schedules. We have also shown that only the parts of such project schedules involving implementation of source code can be produced from the architectural descriptions, activities such as design or analysis must be added from other sources. We also showed that the risk of choosing one alternative or the other was not considered. We therefore propose that risk analysis be included in architectural analysis to make it more explicit (or the opposite, that architectural analysis be used in project risk analysis). This would make it possible to treat risk together with other quality properties and make a conscious trade-off between them. Research in this area will presumably need to incorporate an organizational development and production process model – which would also provide a better basis for time and cost estimation.

7. References

[1] Aiken P. H., *Data Reverse Engineering : Slaying the Legacy Dragon*, ISBN 0-07-000748-9, McGraw Hill, 1996.

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

[3] Bosch J., *Design & Use of Software Architectures*, ISBN 0-201-67494-7, Addison-Wesley, 2000.

[4] Brodie M. L. and Stonebraker M., *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*, Morgan Kaufmann Series in Data Management Systems, ISBN 1558603301, Morgan Kaufmann, 1995.

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

[6] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Evaluating Software Architectures*, SEI Series in Software Engineering, ISBN 0-201-70482-X, Addison-Wesley, 2001.

[7] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Documenting Software Architectures: Views and Beyond*, SEI Series in Software Engineering, ISBN 0201703726, Addison-Wesley, 2002.

[8] Crnkovic Ivica and Larsson M., "Challenges of Component-based Development", In *Journal of Systems & Software*, volume 61, issue 3, 2002.

[9] Cummins F. A., *Enterprise Integration: An Architecture for Enterprise Application and Systems Integration*, ISBN 0471400106, John Wiley & Sons, 2002.

[10] Estublier J., "Software Configuration Management: A Roadmap", In *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, ACM Press, 2000.

[11] Garlan D., Allen R., and Ockerbloom J., "Architectural Mismatch: Why Reuse is so Hard", In *IEEE Software*, volume 12, issue 6, 1995.

[12] Gyllenswärd E., Kap M., and Land R., "Information Organizer - A Comprehensive View on Reuse", In *Proceedings of 4th International Conference on Enterprise Information Systems (ICEIS)*, 2002.

[13] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, 2000.

[14] IEEE Architecture Working Group, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, report IEEE Std 1471-2000, IEEE, 2000.

[15] Johnson P., *Enterprise Software System Integration – An Architectural Perspective*, Ph.D. Thesis, Industrial Information and Control Systems, Royal Institute of Technology, 2002.

[16] Kruchten P., "The 4+1 View Model of Architecture", In *IEEE Software*, volume 12, issue 6, 1995.

[17] Land R., "Applying the IEEE 1471-2000 Recommended Practice to a Software Integration Project", In *Proceedings of International Conference on Software Engineering Research and Practice (SERP'03)*, CSREA Press, 2003.

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

[19] Linthicum D. S., *Enterprise Application Integration*, Addison-Wesley Information Technology Series, ISBN 0201615835, Addison-Wesley, 1999.

[20] Oman P., Hagemeister J., and Ash D., *A Definition and Taxonomy for Software Maintainability*, report SETL Report 91-08-TR, University of Idaho, 1991.

[21] Paulish D., *Architecture-Centric Software Project Management: A Practical Guide*, SEI Series in Software Engineering, ISBN 0-201-73409-5, Addison-Wesley, 2002.

[22] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.

[23] SEI Software Technology Review, *Maintainability Index Technique for Measuring Program Maintainability*, URL: http://www.sei.cmu.edu/str/descriptions/mitmpm.html, 2003.