Architecture-Centric Software Migration for the Evolution of Web-based Systems

Martti Jeenicke

University of Hamburg, Faculty of Informatics,
Software Engineering Group, Vogt-Kölln-Straße 30,
22527 Hamburg, Germany
jeenicke@informatik.uni-hamburg.de

Abstract. This paper presents architecture-centric software migration, an approach which has been used to improve web systems in two projects. Its goal is to increase the potential for evolution by conversion of a web-based system into one that is implemented in a different programming language. In order to reduce the risks and the complexity of the process, it is an agile and evolutionary approach in which only small parts of the web-based information system are changed at once. Thus, while being migrated, the web system can undergo regular further development. In addition to a detailed description of the approach, two migration strategies are introduced and discussed.

1. Introduction

The quality of information systems is undoubtedly important. As described by Offutt [1], it's even more important for web information systems. This is mainly because quality is the only or at least an important factor to get web users back to a web system. For software vendors in traditional system development, a short time to market is often much more beneficial than having a high quality product. Quality can be improved by patches or – even more lucratively – by new releases. Web developers cannot afford to follow this path to evolution. The competitor is often just one click away. Therefore, if no high quality is offered, the users simply won't return. Yet, as described in the next section, many web systems have a quality problem. Some authors, for example, even caution against the possibility of a so-called web crisis comparable to the software crisis of the early 1960s [2]. One solution for avoiding many problems induced by poor quality systems is to replace them with better ones implemented in a higher level programming language that provides developers with better support for meeting the quality requirements. In this way, the preconditions for a successful evolution of the software are enhanced.

This paper presents an approach that was used to achieve this migration for web information systems in two projects without freezing the further development of new functionality. Before presenting two possible migration strategies and the projects in which architecture-centric software migration has been developed, the quality-related problems and peculiarities of web applications will be explored in detail. Then the

underlying definition of migration will be presented after a discussion of related work. Based on this discussion, architecture-centric software migration will be described in greater detail and supplemented by the introduction and evaluation of two strategies for the practical implementation of the migration process. The paper concludes with a summary and an outlook on future work.

2. Peculiarities of Web-based Systems

Software quality can be differentiated by internal and external quality factors [3]. The latter is perceived by the users of a system as e.g. performance, usability etc. The inner quality of a system is made up by the quality of the software architecture, scalability, use of software design principles, readability and understandability of the source code etc. There are dependencies of course. For example, without proper inner software structure, developers won't be able to improve the performance or reliability of an application. Readability is an important precondition for software maintainability and therefore the basis of any bug-fixing and evolution. On the other hand, maintainability is a precondition for the quick release of new versions of a web system and it is also something that web users and customers definitely expect. According to Offutt [1], scalability is the most important quality attribute to be considered, since it influences all other quality attributes. Bad scalability is also a symptom of an underlying lack of evolvability.

Yet, in many web-based systems not enough attention is paid to internal quality due to many facts. One problem that is described quite often, e.g. by Pressman [4] and Murugesan et al. [5], is the fact that web developers are often not appropriately educated.

Another problem is caused by some of the programming languages used for web applications. Common implementation languages for web-based systems are scripting languages such as Perl and Php. Especially these two are very popular among web developers because they offer a large set of pre-defined functions and are fairly easy to learn. Both make it very easy to write a simple web application in a short time. This makes scripting languages also a nice tool for prototyping.

But script-based programming languages have a couple of drawbacks compared to powerful, modern module-based or object-oriented languages. First of all, they don't have good structuring mechanisms for the source code – if they have any at all. Especially the common use of include statements causes readability problems. Include statements are commands for the pre-processor to replace the statement with source code that is stored in a different file. When a file is included depending on a condition, e.g. in an if-clause as it is the case in the source code of CommSy (described below), things get worse. Second, they don't support an advanced type concept. The typing concept of the programming language Php, for example, is not as advanced as that of dynamically typed languages. Type conversion is done on demand, so that it is never clear which type of a variable is in the current state of the program.

Higher software engineering concepts like information hiding or programming by contract are also often not natively supported by the scripting languages used for web

development. If object-oriented concepts are supported, this usually happens at the level of inheritance of source code, i.e. powerful concepts like polymorphism are not really existent.

Another problem of scripting languages is that they don't support testing very well as the testing frameworks are – if there are any – at an infant stage. Testing is a very important prerequisite for the evolution of a software system, as it is a surety for the quality of the system.

The problems of script-based web systems are well-documented in literature. Offutt, for example, states that "many web developers have found that large complex Perl programs can be hard to program correctly, understand, or modify" [1]. Kerer also points out the problems of script-based web applications ([6], page 17).

Thus script language-based programs are unnecessarily complicated and hard to debug and maintain, making them not very usable for large software projects. If they are to be used in those projects, software developers must be highly disciplined as they have to simulate many of the missing features of the higher level languages in order to avoid complicated, incomprehensible and unmaintainable software. This calls for well-educated web developers, which is another problem, as mentioned above. Therefore, a better approach would be to use higher level programming languages for web-based information systems. This also implies that poor systems need to be replaced by new ones implemented in these higher languages.

A peculiarity that web developers are also confronted with when building web-based systems is described by Bleek et al. [7]. Bleek et al. argue that web users expect new versions on a regular basis. Therefore, it is necessary to continue the further development and maintenance of (even poor) web information systems. This also requires that a web system that is in the process of being converted is still usable by its users. I have experienced this first-hand in one of the cases that I will describe in the next section.

Also, development of a web software project never stops – it's hard to come to an end. This means that evolution has to take place in a changing environment. Here we see a difference in comparison with traditional software engineering, where the projects are usually not so dynamic.

3. The Cases

The findings presented in this paper are based on two interdependent projects. The first project was a prototyping process for evaluating and assessing technologies as well as approaches to migration. The second project was subsequent to the first and based on various lessons learned from the first one. The author was highly involved in both projects as both researcher and developer. Empirical data was collected in project diaries, notes, interviews, emails, student assignments and discussions in a web-based community system. I will now briefly describe the two projects.

3.1 A Research Group's Web Site

The first project is the migration of a dynamical web site of the software engineering group at the department of informatics at the University of Hamburg. It is a custom-made and Php-based software system. It was developed in the early days of web development when the language Php was still only available in version 3. Since then, it has been adapted to changing environments a couple of times, e.g. major changes in the programming language occurred. In addition, the system has been bug-fixed and extended. Large parts of the system are not well-documented. Maintenance of the software also became more difficult. A decision was made to replace the system with a new one. It was also clear that the resources for rewriting the system to replace the old one weren't available. Instead it should be migrated in little steps, using an evolutionary approach. Since the group is working a lot with Java, the decision was made to transfer the source code from Php to Java. In order to obtain an understanding of the technologies and possible approaches to developing and releasing the new system, a number of explorative prototypes were built. At the moment, the target software architecture as well as a plan for replacing the system are being developed. However, important findings of the case were the basis of the second project: the prototypical migration of a web-based community system.

3.2 Prototypical CommSy-Migration

Prototypical CommSy migration has been and is being done in a two-semester teaching project at the University of Hamburg. The project is researching a number of questions concerning migration projects of web systems, for example regarding:

- Quality assurance while migrating;
- Appropriate migration strategies for different settings;
- Process implications for the desired target architecture.

Fourteen highly motivated students were involved in exploring and practically using migration approaches. As a practical example for the project we used CommSy. CommSy is a web-based community system which has been in development at the University of Hamburg since 1999. It started out as an endeavour of a couple of students and research assistants, done in their spare time. As the project became bigger and more time-consuming for the people involved, it became clear that it needed to be organized in a different setting. It was then further developed by a publicly funded research project, which allowed for full-time and part-time developers. After this funding ended, CommSy became an open source project in order to provide a frame for interested persons from different organizational settings. Up till then, the focus of development of the system had been always more a usability perspective than a software engineering perspective. As a result, CommSy has a very well-accepted interface design. On the other hand, the system has some drawbacks on the software architecture side as well as a lack of usage of well-accepted principles of good software construction.

As many web-based applications, CommSy is based on open source tools (Linux, MySQL, PHP, Apache). It's written in the scripting language Php, using some of its object-oriented features. With about 90,000 lines of code in about 420 files the system is rather large – at least for a web system. But because of the drawbacks of the scripting language Php, maintenance of the software is now getting more complicated. Therefore, some software design changes became inevitable. This was a good point in time to think about a change in the implementation language. For various reasons, a redesign using Java seemed to be a promising approach.

All this made CommSy an ideal case example for the teaching project. The CommSy development project benefits from the research results as these partly form the basis for decisions on future strategies.

4. Paths to Better Web-based Systems

From a software engineering perspective, there is a need for systems of high quality that are flexible enough to be changed on the architecture level to meet new requirements. This means that the software architecture must be designed not to pose a barrier to change, but to be changeable and developable. Insufficient web systems need to be replaced by better ones that address changeability and are of high(er) quality.

In order to do so, special emphasis must be placed on the software architecture of the system throughout the whole migration process, making use of established construction principles of software and software architecture as well as of design patterns. The new architecture needs to be changeable and developable. Of course the quality requirements of web-based systems must be met. Unfortunately, quality requirements on the software architecture level are not well researched yet. First approaches can be found in [1].

As already described, it is often necessary to switch from a script-based to a more powerful programming language to meet the demands of a higher software quality. The web engineering literature offers no approach for this concern. Very often the approaches describe methods of creating new web-based systems or reference software architectures, e.g. in [8], [9], [10], [11], [12]. Approaches to software migration as described above are – to my knowledge – not discussed. It also looks like the relevance of well-established software design principles in the context of web-based information systems is not discussed.

Using established re-engineering, renovation or restructuring methods seems to be appropriate for the conversion of a web information system.

Arnold [13] describes restructuring as a process in which the structure of an existing software is being changed with the goal of improving one aspect of the system. Examples of such changes are adjustments to the source code to improve readability, to modify documentation or to restructure one component of the system. Refactoring is closely related to restructuring but defined more restrictively. Fowler et al. [14] describe it as "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure". Both

refactoring and restructuring do not change the programming language the system is implemented in. Therefore, both approaches cannot be used to follow the path to a better web information system as described above.

According to van den Brand et al. "the purpose of reengineering or renovation is to study the system, by making a specification at a higher abstraction level, adding new functionality to this specification and develop a completely new system on the basis of the original one by using forward engineering techniques" [15]. But the conversion of a web-based information system should not be done in one big step. We also don't want to add new functionality to it. This should rather be the second step after the conversion of the systems' part that is currently migrated. Therefore, the mainstream reengineering or renovation approaches are not really appropriate.

Closely related to reengineering and renovation is reverse engineering, which is concerned with reinvestigating a system, and design recovery, which also uses other sources than the program code to describe the system at a higher level of abstraction (see [15]). Another term or related approach is architecture reconstruction as described by Bass et al. [16]. As discussed later, all these methods can be useful for the initial steps of a migration if the original developers of the system are not available for questioning.

Related work also exists in the discussion about reengineering patterns. Lloyd et al. [17] for example, describe a number of patterns. But to my knowledge, none of the published patterns seem to address the specific requirements of a migration of a web-based information system as described.

Migration is a term that's being used quite indifferently in scientific information system literature. Sometimes it is used synonymously with the word porting or port even though it has a different meaning. The IEEE [18] defines portability as "… the ease with which a system or component can be transferred from one hardware or software environment to another." Therefore, porting software is the activity in which a system is transferred to another hardware platform or to another operating system. It is not an evolutionary change of the implementation language using small steps. The latter strategy can be better described by the word migration.

5. Software Migration

The term migration originates from the Latin word "migrare" which means "to go from one place to another" (http://www.britannica.com). It's used in sociology and biology to describe a slow but steady movement of a population from one place to another. The interesting fact is that the emphasis is on slow.

Migration is known in many different disciplines, e.g. biology, sociology and geology. In computer science it is used in "data migration" or "hardware migration". The latter means switching from one hardware platform to another one. Data migration is

the process in which the structure of a database is changed and the stored data adjusted to the new structure.

Often the term migration is also used to describe the process of switching from one software system or component to another. One example of this is the transition from commercial software products to open source software.

The unique requirements of converting web information systems possessing the aforementioned constraints are not covered by these explanations of migration.

A change in a programming language almost always implies a change in the architecture of the software system. This is especially true when changing from one type of programming language to another, e.g. from a script language to a higher object-oriented language. Thus the software architecture becomes the focus of concern.

6. Architecture-centric Software Migration

The goal of an architecture-centric software migration process is to gradually convert an information system to one with higher software quality by changing the implementation programming language and the software architecture. Two conditions have to be fulfilled. First of all, the web users' demand (see above) for frequent new software versions has to be met, i.e. the development of new functionality must not come to a halt. Second, the software must remain usable while the process of conversion is taking place. This has been demanded in the case of the CommSy project described above. Both conditions make architecture-centric software migration special. Besides the technical challenges involved in the migration of a life system, the need for a continuing development process for new features presents challenges on the level of process management and communication. Since future developments of the information system under migration are rather unclear, it doesn't make sense to use a big-bang development approach. More suitable in these kinds of situations are agile development methods [19]. One idea of XP as a representative of an agile method, for example, is that the developers are only working on parts of the system that are needed in the present situation. Future developments should be masked out of the current development tasks. This helps to reduce interference between the development process of old parts of the system and the migration process itself, enabling further development or evolution of the web information system while migrating. This is a precondition to satisfy the web users' expectation of frequent new releases which was very important for the prototypical CommSy migration. Here, the development of the CommSy continued in all parts of the system. While changes in the functionality or of a module of the CommSy that was in the process of migration created an overhead in the development process for the new system, the changes were barely recognizable by the developers concerned with migration in other parts of the system. Thus the overall effects of changes to the system were proportionally small in regard to migration. This was also only possible because an approach of using small steps was chosen. The consequence is of course that two development processes take place in parallel: one process

for the old system to meet the web users' demand for frequent new releases, and one that is the actual migration. With this approach to migration, the information system stays usable while being migrated.

Another factor calling for small steps is complexity. High complexity is always a problem when developing software. Therefore, big-bang approaches should be avoided. Particularly large reengineering projects have failed in the past, e.g. the re-writing of a police information system as well as a municipal information system, both of the city of Hamburg. The complexity problems of reengineering projects have also been identified elsewhere. Stevens and Pooley [20], for example, argue for the necessity of an incremental approach. In order to cut complexity, the functionality of the parts of the information system currently migrated should be kept constant.

With the purpose of further reducing complexity, the only visible changes allowed for the system should come from users' requirements. Changes to parts of the system currently in migration should be avoided. As I have experienced in the CommSy project, this is not always possible. Here, while migrating a functionality for making appointments, a new requirement (a team calendar function) of an external user group had to be implemented. Before this requirement was recognized by the developers working on the migration, it had already been implemented by the PHP programmers. Thus it had to be integrated into the ongoing migration process. This example also shows the problem of synchronizing the two development processes inherent to this type of migration. In this case, it was not a major problem as the new program code had not been deployed yet. But synchronizing problems like this can lead to greater problems in terms of the system's quality and should therefore be avoided whenever possible.

In order to maintain the high quality of the software to be migrated, the complexity of the whole process should not only be kept low, but also the communication between the developers in the different processes should be fostered. High attention must also be paid to mechanisms for keeping the software bug-free. Thus architecture-centric software migration relies heavily on automated testing. Testing has to take place both on the internal software level and the level of the user interface. On both levels, a test-first approach is inevitable. The tests for the user interface are more difficult to implement. As experienced in both cases described above, there are many reasons for this approach. The two most important reasons stem from the fact that it is not a "fun job" for programmers to write tests for the user interface of a web application and that the technological problems are quite challenging. The latter is related to the fact that the frameworks for testing web pages, e.g. httpunit, are not mature enough. But tests need to be written for the user interface because the migrated part of the software should look exactly like the old system. This can only be achieved by the use of automated tests. Yet, test-tools like httpunit might have some requirements for the structure of the html-code they are supposed to test. If the old system does not produce the appropriate HTML code, the programmers may have to adapt the old code of the user.

Testing is not only important for product quality. A failed test can also be an indicator that an alteration of the 'old' system has not been discussed between the two

development processes. Thus testing both the old and the new parts of the system and comparing them is important to check if communication between the groups of developers is working or not.

Communication has to be fostered by other means, too. As the developers are separated into two groups (migration and further development), these two groups must find a common language for coordination. The development streams also need to be synchronized as best as possible; the basis of communication here is a common understanding of the software architecture. Discussions on future developments, both in the old and new systems' parts, need to be take place on the software architecture level. Thus the software architecture is the artefact around which the project language forms.

Also, a change of a design decision on the architecture of one group can have tremendous effects on the structure if the system parts are developed by the other group, e.g. in the CommSy project at one point the structure of the session changed. This had a direct effect on session handling in the new system part, as this is the primary mechanism for information exchange between the old and new parts of the system. Also, new functionality for the users can lead to a change in structure of the new parts. At the same time, lessons learned in the migration process can lead to changes in the architecture of the old parts of the system. As the migration is an ongoing (and time-consuming) effort it makes sense to change the old architecture of the system in parts that haven't been migrated yet. This leads to a constant evolution of the whole software system as the migrated parts of the system need to be integrated with the old system elements. This has to be considered while designing the architecture of the new parts of the system. As a consequence, the architecture of the old and the new system parts have to be discussed among the two groups of developers.

Another tool for ensuring the quality of the migrated software is the architecture review. New software parts have to be connected to old parts of the system. Thus the architecture is constantly exposed to changes. This is of course not a problem but a consequence of taking small steps. In a way this also reflects the learning process that is part of any software development. As a consequence, there is no detailed reference architecture for the target system but an intermediate, intended architecture. This means that before the next step in the migration process can be taken, the software architecture of the system – old parts and interwoven new parts – needs to be evaluated. One approach to architecture reviews that has proven to be suitable is the use of tools that analyze the software architecture of a system and attempt to find problems by making use of software metrics.

6.1 Steps of an Architecture-centric Software Migration

The first step of an iteration of an architecture-centric software migration is the decision what part of the system should be migrated first. In order to make this decision, the developers must have a basic understanding of the system to be transferred. As the system will usually still be under development, the easiest way to gain this knowledge

is to discuss the system's structure with the developers that were involved in the original development process. If that is not possible, architecture reconstruction methods as described by Bass et al. [16] should be used to analyze the system.

The architecture of an information system is usually based on a reference architecture. Common types also found in information systems are module-based, client-server, layer structured or peer-to-peer. The goal of architecture-centric software migration is to replace web information systems with ones of a higher quality. But it is often very difficult to identify a reference architecture in low quality web information systems, as there usually exists a lack of structure. Sometimes these systems are not even based on reference architectures. But even the most chaotic systems actually have at least an identifiable modularization on the level of single web pages that are displayed. In many script-based systems, for example, a set of viewable web pages logically belonging to one functionality are generated by source code that is located in a single file.

After the decision on what to migrate has been made the developers need to choose a strategy to achieve the goal for the iteration. Both questions are interdependent. There are as many options on what part of the system to work on first as there are identifiable parts. Yet only two starting points seem to be reasonable: The first option is to begin migration with the simplest part of the system in order to explore the particularities of the system to be migrated. Further migration iterations of more complicated parts of the system can then profit from these experiences. This approach was chosen in the prototypical CommSy migration. The learning curve of it is valuable not only for a prototypical approach, but also for a full migration. The other option is to start migration in the part of the system that has the most quality-related problems, i.e. the part of a migration that brings the greatest benefit to the web information system. But starting with the most complicated parts of the system is risky and calls for a very experienced migration team. This approach has been used in the university web site project described above.

Once it is clear what to start with, the decision on the strategy of how to accomplish the migration must be made. What we have learned from the two cases is that there are two feasible ideal approaches: horizontal and vertical migration. Both strategies do not try to achieve the transition by a line to line translation. Instead, architecture-centric migration keeps the architecture in the focus of all design decisions.

6.2 Horizontal Migration

Horizontal migration is only possible if a layer or module-based layer structure (but not based on web pages) of the system to be migrated can be identified. If this is the case, one layer can be replaced by new software. Challenging is the integration of the new part into the old system, as the mechanisms for interaction between both programming languages have to be integrated on both. Thus the technological overhead is tremendous.

The easiest way to integrate new software into the system is to use the old structure of the system as a wrapper for the new software. This way only the wrapper is affected by the new system. The disadvantage of this method is the overhead at runtime and the fact that the wrapper is only easy to maintain as long as only one layer of the system is migrated. After another one has been migrated, the management of which function to call from where (old or new) gets fairly complicated.

Even though it depends on the old systems' structure, the probability that a horizontal approach will lead to many connection points of the old and the new system parts is quite high compared to the vertical approach described next. This leads to a high coupling of the system modules which must be avoided from a software engineering standpoint. It also increases the need for communication between the two developer groups and makes the software architecture less flexible in the migration process. Its advantage is the flexibility to migrate very small parts of the software by using a wrapper structure.

6.3 Vertical Migration

In vertical migration, one aspect or functionality of the information system is converted completely, i.e. all necessary parts from the user interface to the database connection must be rewritten in the new language. If a layered structure is identifiable, this means that the functionality of the old system must be moved to the new one across all layers. The above-discussed systems that have a file-based module structure can only be migrated with a vertical approach by replacing file by file of the old system with the new functionality and software architecture.

A vertical migration implies a lower connection between the old and the new parts of the system, i.e. the positive effect from a software engineering perspective is a loosely coupled system. This leads to a lessened need for synchronization of the two development processes compared to the vertical approach. It also makes the architecture more flexible.

The disadvantage of this approach is that it is not unusual that the vertical approach cannot be done purely. Most software systems' functionality cannot clearly be divided into software parts that are responsible for just one function. Thus in a vertical migration it might be necessary to either do calls to not yet migrated parts of the system or to implement the necessary – and only the necessary – functions of needed system parts within the iteration of the vertical migration. The first has the disadvantage of being technologically challenging, whereas the latter poses the problem that code is duplicated until the whole system has been migrated. Also, duplication is in fact not a purely vertical migration, but one that also has horizontal aspects. In this regard, it is more a vertical puzzle-piece approach.

In the case of the prototypical CommSy migration, we used a vertical approach. It has proven to be quite easy to integrate the rewritten part of the system into the old one by simply using web links. The CommSy has different modules for different functions, e.g. a discussion module, a module for news and another one for a description of users. These modules are connected with each other using web links, i.e. a user view-

ing a page in one module can jump to another module with different functionality by clicking on a link. However, all requests for web pages are managed by a central entry point. In the project, we decided to migrate a module for appointments from PHP to Java. After the module had been rewritten in Java it needed to be woven into the existing system. The only step necessary to achieve this was to build a central mechanism into both the old and the new system that ensures that the right module – old or new – of the system is called.

We decided not to use the horizontal approach because the mastering of the technological barrier took more time than would have been appropriate for a teaching project.

The next step of the architecture-centric software migration is to actually do the migration and release the new system after thorough testing. Then the whole process starts all over again with identifying what to migrate in the next iteration and the selection of the strategy. There is no necessary dependency on a strategy chosen in past iterations. This leads to a combination of vertical and horizontal migration. As a result of more and more iterations of this process, the system slowly grows towards a new architecture and will be implemented in a new programming language.

6.4 The Challenges

Architecture-centric software migration is not without risk. Besides the challenge of having to manage a migration process parallel to the 'normal' development process, there are also a number of technological challenges.

One of these challenges is the interaction of different programming languages. There are several ways in which two different programming languages can interact. For example, for some languages there are mechanisms that allow calling methods or functions of one language from another. But these mechanisms are technologically challenging, not available for all languages and imply a technological overhead that can have negative effects on the software architecture and the performance of a system. Web-based systems also offer a different way for switching between parts of a system: the links or URLs. They offer a way to call functions that are implemented in a different programming language. But the destination of a web link is usually a web page. Therefore this approach is rather useful when doing a vertical migration, e.g. on a page-based system.

Another challenge is the data exchange between programming languages within a web-based information system. One characteristic of web applications is that the state of the application is stored in a session. Sessions of different programming languages are usually not compatible. Therefore, when doing an architecture-centric software migration of a web application, a mechanism for exchanging and synchronizing data stored on the session needs to be provided.

Since the session data of Php and Java are not compatible, a mechanism for synchronizing the data was also built during the prototypical CommSy migration. This mechanism causes an overhead whenever a page is called. But this overhead only

exists until the migration has been completed. It works like a detour when road construction is preventing traffic from flowing normally.

7. Conclusion

The presented approach of architecture-centric software migration of web-based systems addresses the peculiarities of web information systems. It allows for a stepwise agile migration of systems of a bad quality to better systems that are implemented in a different programming language. After a successful migration, the web-based system is ready for further evolution. The two presented strategies for migration have been proven to work in the two cases presented. The description of the advantages and disadvantages of the strategies is a good basis for managing migration processes of web-based information systems.

The use of small steps within an evolutionary process reduces the risk of a migration and makes the change of language possible and predictable. Automated testing and a test-first approach also help to improve the quality of the migrated system. The focus on the change of the software architecture leads to high quality software designs that meet the quality requirements of web information systems as called for by Offutt [1].

Therefore, the architecture-centric software migration is a promising way to improve the quality of web-based systems and a way to improve the potential for software evolution.

8. Future Work

Even though the architecture-centric software migration of web-based information systems already works quite well, a lot of research is yet required. First of all, work on quality attributes of web-based information systems needs to be extended. Vertical and horizontal strategies also must be further explored. Especially the consequence of using both approaches in a migration project is of interest, as it might be a way of accelerating the speed of software migrations. Further research on architecture-centric software migration involving other projects is necessary. This should then lead to research on how software engineering and re-engineering techniques can be used in or adapted for software migration.

Another point for future research is how the communication between the two developer groups and the synchronization of the processes can be supported and managed.

Finally, the architecture-centric software migration must also be tested in other areas of information systems.

9. References

1. Offutt, J.: Quality Attributes of Web Software Applications. IEEE Software: Special Issue on Software Engineering of Internet Software, vol. 19, no. 2, (2002) 25-32.
2. Ginige, A. and Murugesan, S.: Guest Editors' Introduction: Web Engineering – An Introduction. IEEE MultiMedia, vol. 1, no. 8, (2001) 14-18.
3. Meyer, B.: Object-oriented Software Construction. 2nd ed. Prentice Hall, Upper Saddle River, NJ, 1997.
4. Pressman, R.S.: What a Tangled Web We Weave. IEEE Software, vol. 17, no. 1, (2000) 18-21.
5. Murugesan, S., Deshpande, Y., Hansen, S., and Ginige, A.: Web Engineering: A New Discipline for Development of Web-based Systems. First ICSE Workshop on Web Engineering (WebE-99), Los Angeles, USA, (1999) 1-9.
6. Kerer, C.: XGuide – Concurrent Web Development with Contracts. Technische Universität Wien, Fakultät für Technische Naturwissenschaften und Informatik, Technischen Universität Wien, Austria, (2003).
7. Bleek, W.-G., Jeenicke, M., and Klischewski, R.: e-Prototyping: Interactive Analysis of Web User Requirements. Journal of Web Engineering, vol. 3, no. 2, (2004) 77-94.
8. Cecchet, E., A. Chanda, Elnikety, S., Marguerite, J., Zwaenepoel, W.: A Comparison of Software Architectures for E-business Applications. Rice University Computer Science Technical Report TR02-389, (2002).
9. Kappel, G.: Web Engineering: Systematische Entwicklung von Web-Anwendungen. dpunkt-Verlag, Heidelberg, (2003).
10. Romberg, T., and Anastopoulos, M.: Referenzarchitekturen für Web-Applikationen. FZI internal, Projektbericht App2Web, FZI/Fraunhofer IESE, (2001).
11. Pipka, J.U.: Test-driven Web Application Development in Java, Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002, Springer, Heidelberg, Erfurt, Germany, (2002) 378-393.
12. Kurniawan, B., and Xue, J.: A Comparative Study of Web Application Design Models Using the Java Technologies. The Sixth Asia Pacific Web Conference – APWeb 2004, Hangzhou, China, (2004) 711-721.
13. Arnold, R.: Software Restructuring. Proceedings of the IEEE, vol. 77, no. 4, April 1989, (1989) 607-617.
14. Fowler, M., K. Beck, et al.: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Boston, (2001).
15. van den Brand, M., Klint, P., and Verhoef, C.: Reverse Engineering and System Renovation – An Annotated Bibliography. ACM SIGSOFT Software Engineering Notes, vol. 22 , no. 1, (January 1997) 57-68.
16. Bass L., Clements, P, and Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading, Mass, 1998.
17. Lloyd, A.D., Dewar, R., Pooley, R.: Legacy Information Systems and Business Process Change: A Patterns Perspective. Communications of the Association for Information Systems, vol. 2, no. 24, 1999.
18. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers, New York, NY, (1990).
19. Ambler, S.W.: Duking It Out. Software Development 10, July 2002.
20. Stevens, P., Pooley, R.: Systems Reengineering Patterns. Proceedings ACM-SIGSOFT, 6th International Symposium on the Foundations of Software Engineering (1998) 17-23.