Renaissance: A Method to Support Software System Evolution

Ian Warren and Jane Ransom
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK

Email iw | bjr@comp.lancs.ac.uk

Abstract

Legacy systems are often business critical and are associated with high maintenance costs. In this paper, we present an overview of a method, Renaissance, which aims to manage the process of regaining control over such systems. Renaissance supports system evolution by first recovering a stable basis using reengineering, and subsequently continuously improving the system by a stream of incremental changes. In both cases, the extent of evolution is determined by a phase which takes account of technical, business, and organisational factors. Renaissance defines a process framework, a predefined number of evolution strategies, an information repository, and a generic set of personnel responsibilities. The method can be tailored to the needs of particular projects and organisations, and it is not prescriptive of particular tools and techniques.

Key words: evolution, reengineering, method, legacy system, incremental, maintenance.

1. Introduction

The need for change in software systems is now well understood. Despite being a topical subject, highlighted by Year 2000 and European single currency projects, change is inherent in software systems. Factors which initiate change include changes in user requirements, the emergence of new technology, changes in business goals, and the need for organisations to exploit new opportunities. Unless systems can be changed to respond to their changing environments, they will progressively become less useful [1].

The developed world has invested heavily in software systems. However, many of these systems are what we define as *legacy systems* [2]. A legacy system is an old system that remains in operation within an organisation. Many legacy systems are business-critical and are expensive to maintain. The reasons for high maintenance costs include short anticipated system lifetimes, the immaturity of software engineering practice at the time of their construction, and the sacrifice in maintainability needed to satisfy other constraints. Memory, for example, was a particularly expensive resource at the time when many of today's legacy systems were developed. To reduce system costs, software was often designed to work with limited memory rather than to ensure maintainability.

Organisations which depend on legacy systems face a dilemma [3]. Continuing to maintain the system is expensive and may prove ineffective in accommodating necessary changes. Where a system suffers from severe structural decay, for example, it can be difficult to integrate new functionality. Organisations may consider replacing the system, but the costs of developing and deploying a new system may be prohibitively high. Furthermore, replacing a legacy system incurs the risk of losing business critical information; this in itself can prove fatal.

Reengineering offers a compromise between continued maintenance and replacement of a legacy system. Like maintenance, the starting point for reengineering is an existing system and not a new development project. Unlike maintenance, however, reengineering usually involves improving the system in some way, with the aim of reducing the costs and risks associated with subsequent evolution [4]. Studies have shown that reengineering, where appropriate, is generally more cost effective and less risky than developing a new system [5].

While developments in reengineering contribute towards transforming legacy systems to evolveable systems, managing the process of system change has largely been ignored [6]. In this paper, we provide an overview of a method to support software system *evolution*. We use the term evolution to mean a controlled approach to system change. The Renaissance method [2, 7] is the result of the European Esprit project RENAISSANCE, which was commissioned to address the problems of managing system evolution. Renaissance provides support for managing evolution projects, from their conception though to deploying an evolveable system.

2. Objectives

Providing a controlled approach to system change essentially means reducing the costs and risks associated with change. With these aims in mind, we have identified four key requirements of a method to support system evolution (Table 1).

R1.	The method should support incremental evolution.
R2.	Where appropriate, the method should emphasise reengineering, rather than system replacement.
R3.	The method should prevent the legacy phenomena from reoccurring.
R4.	It should be possible to customise the method to particular organisations and projects.

Table 1 Method requirements

The first requirement aims to reduce the risks and spread the costs associated with system evolution. A stream of incremental changes to a system involves less risk than a single large reengineering effort or a new development project since developers can exploit the feedback which is available after each increment. In addition, incremental evolution enables costs to be distributed over time [5].

In many cases, reengineering would reduce evolution costs and risks. However, reengineering is not always appropriate. Requirement R2 addresses this issue; the method should provide guidance on when to reengineer a system and when to replace it.

To reduce the costs and risks of evolution in the long term, a method which supports evolution should do more than support a single evolution project. The method would offer significantly greater value if it regained control over a legacy system for the remainder of its useful lifetime. Requirement R3 addresses this need.

Requirement R4 recognises that evolution projects are diverse and that organisations have their own established processes and tools. To be successful, the method should be sufficiently flexible so that it can support this variety, while at the same time effectively manage evolution. Furthermore, the method should not impose an overhead which where the cost of following the method is too high for the return on that investment.

3. The Renaissance method

With Renaissance, we propose a two-stage process for transforming legacy systems to evolveable systems (Figure 1). First, a stable basis should be established from where the legacy system can be evolved cost effectively. This stage involves strategic planning, which may result in reengineering the legacy system, or in extreme cases, replacing it. Once the system has been transformed, stage 2 applies a programme of continuous improvement to the system for the remainder of its useful life. Continuous improvement is a fundamental property of evolveable systems which preserves their evolveability.

Stage 1 is generally more challenging and requires more effort than the second stage. One reason for this is that recovering a stable basis often involves a system modelling exercise. Modelling can be an expensive task, but it helps developers to understand the system, which is essential for making informed system evolution decisions. The first stage may also be expensive because of a need to migrate from obsolete technology to modern well-supported technology. This may involve both hardware and software costs, labour-intensive system transformation activities, such as migrating data, and system restructuring.

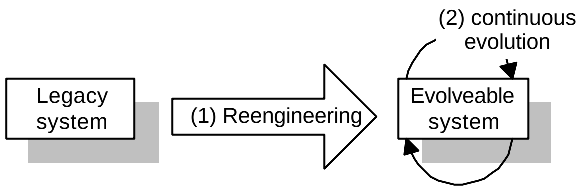

Figure 1 The Renaissance approach

The second stage assumes a system whose evolveability is assured, given anticipated changes in its environment. Stage 2 generally involves a stream of evolution iterations which do not require radical reworking of the system. Since the system is evolveable, the costs and risks and associated with each change can be predicted with considerably better certainty than the costs and risks associated with maintaining many legacy systems.

The Renaissance method comprises a classification of evolution strategies, a process framework, an information repository, and a set of responsibilities to be met in a typical evolution project. Each of these elements can be tailored to fit particular project and organisational factors. For example, a project to regain control over a legacy system (a stage 1 project), will invariably require more emphasis on long-term planning and reverse engineering than a stage 2 project. This emphasis means focusing on particular Renaissance activities, documents, and responsibilities.

3.1 Evolution strategies

An evolution strategy describes a particular form which system evolution might take. We define six strategies which span continued maintenance, reengineering, and system replacement. Since reengineering is a general term, we define four particular classes of reengineering. Table 2 presents these strategies, in general, in terms of increasing cost, benefit and risk.

For many evolution projects, a hybrid evolution strategy is often appropriate. For example, an organisation may initially decide to upgrade a system's user interface from a text-based interface to a GUI solution. Subsequently, after users gain experience with the system and when sufficient budget is available, the system may be migrated to a client/server infrastructure. In this case the strategy can be implemented in two increments.

Continued Maintenance	The accommodation of change in a system, without radical change to its structure, after it has been delivered and deployed.
Revamp	The transformation of a system by modifying or replacing its user interfaces. The internal workings of the system remain intact, but the system appears to have changed to the user.
Restructure	The transformation of a system's internal structure without changing any external interfaces.
Rearchitecture	The transformation of a system by migrating it to a different technological architecture.
Redesign with Reuse	The transformation of a system by redeveloping it utilising some legacy system components.
Replace	Total replacement of a system.

Table 2 Evolution strategies

Continued Maintenance is often the most cost-effective strategy, particularly when the system is well-documented, when there is no shortage of experienced maintenance personnel, when the system is in good technical condition and when it is able to accommodate forecasted changes in requirements. For many legacy systems, however, Continued Maintenance is inappropriate.

Where estimates show reduced maintenance costs following reengineering, reengineering may be a better strategy. In addition, where a system is in poor technical condition, or where it is dependent on obsolete technology, reengineering may also be appropriate. System replacement should only be considered when reengineering solutions are not feasible or where they do not significantly reduce cost and risk.

The Revamp strategy is appropriate where the motivation for change is to improve the presentation of the system in terms of its user interface. Middleware tools exist to interpret the character stream of a text-based user interface and generate GUI events [8]. Where application logic and the user interface are loosely coupled, use of such middleware products often requires no change to the source code of the legacy system. While the Revamp strategy gives the illusion of a new system, it does not increase the system's evolveability since any problems with its data or logic structures remain.

Restructure is the inverse of Revamp. A restructured system should be more evolveable since the objective of restructuring is to simplify the system's logic and data structures. This makes the system easier to understand and evolve. Restructuring a system typically involves improving the flow of procedural components, removing redundant code, conforming to coding standards, and reducing component dependencies. Like Revamp, there are several tools which can be used to automate system restructuring. There is no immediate return on the investment in restructuring, but medium-term evolution costs should fall because of the system's improved evolveability.

Rearchitecture is a more radical strategy than Restructure. Rearchitecturing involves transforming a centralised architecture to a distributed model. Hardware and software rearchitecturing are generally interdependent. Rearchitecturing is more expensive and carries greater risk than restructuring, but its benefits are greater. A rearchitectured system is likely to be evolveable in the long-term because it runs on modern well-supported hardware. In addition, modern client/server, object-oriented, and component-based technologies typically conform to interoperability standards and are generally more responsive to change.

Redesign with Reuse is a variation on Rearchitecturing which aims to preserve the knowledge and business rules that are embedded in legacy system components by integrating them within a new architecture. In [9] we describe how both data and procedural components can be encapsulated for use in a new architecture.

3.2 Process Model

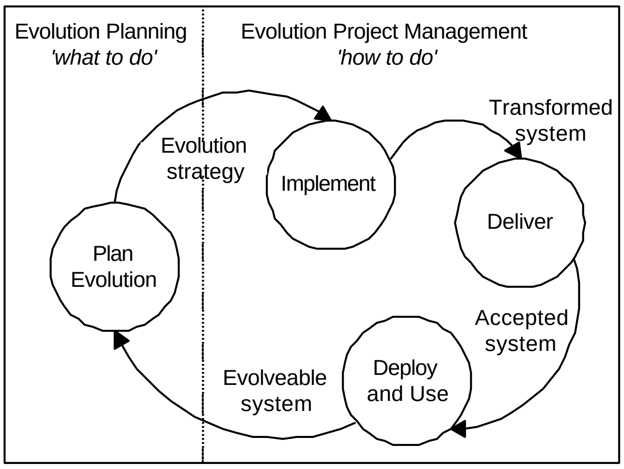

Figure 2 Abstract process model

Figure 2 shows how the process model is split into four high-level phases and Table 3 identifies key activities for each phase. The entry phase for any evolution project is Plan Evolution, which addresses the system's long-term future. Plan Evolution involves assessing the current system from technical, business, and organisational perspectives with a view to develop an evolution strategy.

Phase	Activities
Plan Evolution	Calibrate method Assess system Develop evolution strategy
Implement	Plan evolution project Design, transform and test system Prepare environment
Deliver	Migrate data Install system Train operators
Deploy and Use	Changeover system Evaluate system Document environment changes

Table 3 Key activities

The decision reached in the Plan Evolution phase determines whether the remaining three phases should be followed. Where a system is found to have no useful future, there is no need to continue with the method. In other cases, phases Implement, Deliver, and Deploy and Use support the tasks of managing the evolution project and making the transition between the current and transformed systems according to the evolution strategy.

Plan Evolution starts with a calibration activity where the method is customised to take account of particular project and organisational factors. In particular, this activity involves deciding which phases of the method's process are needed, determining the level at which to perform activities within these phases, selecting techniques and tools with which to perform the activities, and identifying individuals who will meet the necessary responsibilities.

In general, we recommend techniques and tools to support method activities. In [9] for example, we outlined an attribute-rating scheme for assessing legacy systems for fitness for evolution. In [10] we describe how the UML can be used to develop system models. However, we expect that practitioners will override some of our suggestions in favour of their established procedures.

Once calibrated, the rationale for using Renaissance and any project constraints should be recorded in the information repository. These pieces of information contribute towards the development of a system's evolution record. Such a record assists developers to make informed decisions in subsequent evolution iterations [4]. Renaissance can be used to manage many evolution scenarios, such as: systems whose maintenance costs are too high; organisations which intend to pursue new markets and need to determine how their systems will need to change; and situations where systems cannot be effectively maintained to integrate new requirements.

Project constraints time, budget, deployment, and technology constraints. To illustrate a technology constraint, in one case study, we saw that an evolution project was initiated because of senior management's wish to migrate their mainframe system to a distributed client/server system. In this case, the motivation for prescribing particular technology was to achieve a standard technology across a large enterprise.

The key to effective evolution planning is a thorough assessment of the legacy system [9]. Assessing a system means calculating measures for the system's technical quality and business value, and taking account of any organisational factors which might affect an evolution project.

The technical quality of a system can be measured by assessing its software and hardware. Software can be classified as application software or support software. The latter includes operating systems, 4GLs, TP monitors and compilers. To measure a system's technical quality, we propose that attributes should be selected for components and assigned values. One attribute that we suggest for hardware and support software components is vendor. If a component's vendor is no longer in existence, we would rate vendor low. An improved value would be given where the vendor no longer exists, but a third party supports the component. A high value would be awarded in cases where the vendor exists and its future seems assured. We described this technique in more detail in [9].

In the absence of reliable documentation and individuals who understand how a system has been implemented, it will be necessary to build context models to support the assessment exercise. In [10] we describe how to build both context and technical models of legacy systems using the UML. A context model describes a system at an abstract level and from four views: business, functional, structural, and environmental. The business view captures a system's support for its business process. The functional view describes the services that implement the business process. The structural views gives an overview of the main configuration elements of the system and the environmental view shows the main physical devices.

To determine the business value of a system, its organisation's business goals should be captured. Business goals generate tomorrow's system requirements and they heavily influence the choice of evolution strategy. For example, an organisation may have a strategic plan which makes a system redundant. In this case, continued maintenance of that system is likely to be the only cost effective evolution strategy. In other cases, business goals may show that the system's data and services are critical to the organisation's continued operation. Here, the system must survive, in some form.

Organisational factors span the organisations which are involved in operating the system, and those that are responsible for evolving the system. For the former, the organisation's attitude to change could affect the success of an evolution project. Where an organisation appears to resist change, for example, an incremental evolution strategy may reduce the probability of users rejecting the evolved system. For the latter, the availability of maintenance personnel who are experienced with a particular legacy system is likely to affect the degree of modelling needed to understand the system.

Based on the results of assessment, an evolution strategy can be developed. For each evolution strategy introduced in Table 2, we provide guidance on its applicability. For example, each strategy is associated with particular costs and risks; in Table 4 we point out possible risk factors. In [11], we elaborate on risk management and cost estimation in the context of evolution projects. In many cases, a number of evolution strategies will be identified as candidates. To pick the most appropriate strategy, an exercise which considers the relative costs, benefits, and risks of each strategy should be performed.

	Continued Maintenance	Revamp	Restructure	Rearchitecture	Redesign with Reuse	Replace
Lack of system knowledge	✓	✓	✓	✓	✓	✓
Lack of experienced maintenance personnel	✓	✓				
Poor documentation	✓	✓	✓	✓	✓	✓
New technology skills shortage				✓	✓	✓
Legacy technology skills shortage	✓	✓	✓			
Errors introduced during evolution			✓	✓	✓	✓
Technology immaturity				✓	✓	✓
Loss of embedded business rules						✓
System will not meet evolution requirements	✓	✓				
Obsolete operational environment	✓	✓	✓			

Table 4 Evolution strategy risk factors

Phase Implement begins with planning the evolution project which will implement the chosen evolution strategy. Project planning has more in common with traditional development project planning than the strategic planning of the first phase. As such, project planning involves allocating resources, managing risk, estimating costs, and scheduling.

However, since evolution projects involve transforming an existing system, they raise a number of additional issues not found in new-development projects. Evolution project planning should also tackle data migration, system deployment, managing an incremental system transformation, and integrating legacy system components with new technology. Since we are constrained by space in this paper, we introduce data migration and system deployment. In [2, 11], we describe all these issues in greater depth.

In many cases, especially involving commercial systems, the deployment period should be as short as possible so as to minimise disturbance to the organisation which relies on the system. The evolution strategy dictates, to a large extent, how a system can be deployed. For strategies which do not involve radical reengineering or replacing the system, the system can often be deployed incrementally. With restructuring for example, as components are restructured, they can be integrated in a new version of the system to be released and deployed. In this way, the legacy system is transformed through a series of increments. In many cases, deploying the system incrementally can be performed without affecting system users.

For more challenging reengineering projects or where the legacy system is to be replaced, the legacy system typically has to be decommissioned before the new system can be deployed. In cases where the replacement system is built from fundamentally different technology to the legacy system, or where it implements a revised business process, this is the only practical solution. However, the drawback of this approach is the inevitable disruption and loss of service experienced by the organisation which operates the system.

Where a break in system service cannot be avoided, the deployment period can be minimised by effective project planning. Any activities which can be completed before deployment should be scheduled so that they are finished before the legacy system is taken out of service. Of those activities which Renaissance defines for the latter three phases, designing, transforming and testing the new system can all be completed prior to deployment. In addition, once the system has been acceptance tested, operators can be trained before the transformed system is deployed.

In cases where the transformed system involves new hardware or a new architecture, Renaissance defines an activity to prepare the new environment. Where possible, this activity should also be completed before deployment. There are occasions, however, where this is not feasible, such as where space does not permit new hardware to coexist with old hardware or where the disruption caused by the preparatory work would be excessive.

One activity which can be particularly challenging when reengineering data processing systems is data migration. Renaissance defines an activity to deal with this process which covers identifying data which must survive evolution, converting data to new formats, and integrating legacy data with the transformed system.

Several activities depend on the availability of migrated data. Planning is thus important to ensure that sufficient data has been migrated in time for dependent tasks. System testing, for example, can exploit the availability of real data to provide a realistic test environment if data has been migrated beforehand. In practice, however, many data cannot be migrated until the deployment period. To provide guidance on when particular data can be migrated, we distinguish between static and dynamic data.

Static data generally equates to an application's master data, and dynamic data relates to transaction data. To illustrate this distinction, in a retail application static data would include supplier, customer, and product-type information; and dynamic data would constitute purchase orders, sales, and deliveries. Static data does not change often and it can be migrated prior to deployment. During the period between migrating the data and deploying the target system, any changes to static data on the legacy system should be recorded and used to maintain consistency between the live and migrated data. Given the volatile nature of dynamic data, migrating it should be delayed until deployment.

Renaissance does not finish when the transformed system has been deployed. Phase Deploy and Use aims to preserve the evolveability that an evolution project has granted a system. The final phase includes activities to evaluate the system, with the view of capturing new requirements to be considered for subsequent evolution iterations. Any new requirements in addition to changes in the system's environment should be recorded to help maintain the system's usefulness within its operational organisation.

3.3 Information Management

Renaissance defines a structured document repository for managing information which can be used throughout a system's evolution. Table 5 shows how we classify information, and how these classes are related to phases in the method's process. Each class contains a set of logically-related documents. These documents should not be discarded after an evolution project, but instead, they should persist so that developers can use them to make informed decisions when evolving the system in the future.

Business documents describe the organisation's business goals and the business process supported by a system. System documents include an assessment report which contains measures of the system's business value and technical quality. In addition, this folder contains any documentation which has been collected for the system, and the results of any modelling.

Based on the work carried out in the Plan Evolution phase, the Evolution Strategy documents describe the possible evolution strategies for a system. Where there is more than one strategy to consider, these documents should also describes the relative costs, risks, and benefits of each strategy and the rationale for choosing a particular strategy. In this way, these documents record evolution rationale, which adds value to subsequent evolution projects involving the system.

Document class	**Plan Evolution**	**Implement**	**Deliver**	**Deploy and Use**
Business	✓	✓	✓	✓
System	✓	✓	✓	✓
Evolution Strategy	✓	✓		
Project Management		✓	✓	✓
Test		✓	✓	

Table 5 Repository structure and usage

Project Management documents record the details of project planning and deployment planning. Where an incremental evolution strategy has been developed, these documents describe the schedule and activities needed to produce a stream of system increments. Test documents include the testing strategy used for the project, test data and the results of testing.

The repository can be realised as a database management system, or as a collection of electronic files in a directory structure, or as paper documents. In practice, it is likely that some combination of these media will be used. For old legacy systems, any documentation that has survived may be paper-based. As the method is followed, resulting documents, such as UML models or project plans, will almost certainly be generated using tools. Depending on the effort used to follow the method, a context model could be a short word processed description, or it could be multi-viewpoint UML model.

Once information has been added to the repository, subsequent activities and evolution projects involving the system can use it. Table 5 shows the relationship between the process model and the repository. For example, a description of the business process, a Business document, is created during the Plan Evolution phase. It is available to be used in the remaining phases to generate functional requirements, to support system testing, and to help with evaluating the transformed system. In other cases, documents developed in one phase may be refined in subsequent phases. For example, context models developed in phase Plan Evolution can be elaborated into technical models to support system design and transformation in phase Implement.

3.4 Responsibilities

Renaissance defines a set of responsibilities which need to be met in a typical evolution project. This set of responsibilities may be incomplete for some projects, and in other cases it might include superfluous responsibilities. Identifying responsibilities and assigning them to individuals is part of project planning. Given a particular project, an individual might meet several responsibilities, and conversely, several individuals might meet a single responsibility.

We classify the responsibilities in two groups: those related to the organisation which operates the system, and those which are technical. The former group includes individuals who understand the system's role within their organisation. Such individuals are often senior personnel and should be able to answer questions concerning the system's purpose and functionality. In addition, a range of system users, including those who physically operate the system and those who benefit indirectly from its services, should be identified. Collectively, such individuals help technical personnel to understand the behaviour of the system.

Technical responsibilities represent the computing professionals who will be involved in the evolution project. One valued technical responsibility, but one which is not always possible to meet, is legacy system developer. This responsibility requires engineers who have developed a good working knowledge of the system from a technical perspective. Unfortunately, in many evolution projects, individuals who could meet this responsibility are no longer available. In this case, system models have to be built to gain the necessary understanding of how the system has been implemented and changed.

Other technical responsibilities include modellers, quality engineer, developers, and project manager. Individuals with expertise in modelling systems built from both legacy and modern technology can meet the modeller responsibility. The understanding gained from well constructed models provides a good foundation on which to base project decision-making. As with new development projects, individuals need to be responsible for quality assurance and project management. Depending on the evolution strategy, developers may need experience with legacy technology.

4. Evaluation and Discussion

During the Renaissance project, industrial partners were involved in evaluating the method. Each partner used applications with different technical, business, and organisational properties. Based on their findings, the method was refined. Since further evaluation, we have started to collect initial results (Table 6).

Strengths	Weaknesses
Well-defined process	Adoption overhead
Application assessment method	Overhead for small projects
Evolution strategy selection process	
Customisability	
Protection of investment in current systems	
Business-driven nature	

Table 6 Initial evaluation summary

The process framework was found to be easy to follow with a logical flow of activities which mapped well onto the concrete steps taken in evolution projects. The lack of detailed project management support was viewed positively, since partners used established procedures of their own. For example, Renaissance was integrated with ISO9000-certified project management processes without difficulty.

Requirement R1, the need for the method to support incremental evolution (Section 2), was satisfied by the method with the positive consequence that project complexity was reduced. The evaluation confirmed that risks can be reduced and costs distributed when a project is managed incrementally. Furthermore, in one case, incremental reengineering was exploited to manage staff training so that engineers were prepared for using new technology needed to implement particular increments. In this way, existing staff can be retained and allowed to develop new skills which are needed for a system's evolution.

The evolution strategy selection process, based on the predefined evolution strategies, their costs, benefits, and risks, also proved useful. Evaluators found this advice useful when determining whether to reengineer or replace their systems (R2). In practice, hybrid evolution strategies were used, where different evolution strategies are applied to different components within a system.

Throughout evaluation, the ability to customise the method to account for particular organisational and project factors (R4) was seen as essential. For example, some evaluation work used the method to conduct a feasibility study. In this case, only the Plan Evolution phase, a subset of documents, and a subset of responsibilities were necessary to carry out the study. Used in this way, the method can support an exercise which aims to prioritise an organisation's applications as candidates for evolution. Where an organisation has several applications, it can quickly determine which systems need to be addressed first [12].

Renaissance allows the inversely proportional relationship between cost and risk to be exploited. For example, to gain an approximate measure of a system's technical quality, the assessment process can be performed at a high level with little effort. For a measure which is associated with more confidence, the process can be performed at a more detailed level. In other cases, a ball-park figure for evolution costs might be sufficient. Here, a technique based on expert judgement could be used rather a more accurate, but more time consuming technique. Choosing the level at which to perform an activity helps to reduce the overhead of using Renaissance.

However, the evaluation revealed that the overhead of adopting Renaissance for the first few projects is high. Initially, evolution costs were estimated to be higher when using Renaissance than if it were not used. Having gained experience and familiarity with using Renaissance, evaluators found the overhead of using the method to drop significantly. It is to be expected that adopting a new non-trivial method involves a learning curve. Furthermore, the costs of using Renaissance should be put in context with its benefits and the reduction in risk through using it.

In addition to the overhead of learning how to use Renaissance, another weakness reported was the overhead imposed for managing small projects. Once over the learning curve, evaluators found that the method was better suited to managing medium-to-large projects than small projects. We are currently considering a lightweight version of Renaissance, with a reduced number of activities, documents and responsibilities.

Method requirement R3, introduced in Section 2, concerns the need for the method to preserve the evolveability of a transformed system for the remainder of its useful life. Whilst it is difficult to evaluate whether this requirement has been satisfied in the short term, the Renaissance approach has been designed to account for it. With Renaissance, a system's evolution is managed by a stream of evolution increments. This is analogous to the Kaizen approach to business process change [13]. Kaizen reduces the risks of change by managing change as a series of conservative improvements to the process. However, this approach requires that the process is in a state where it can be effectively improved. For software systems, this means recovering a stable basis using reengineering.

5. References

[1] Lehman, M. M. and Belady, L. *Program Evolution: Processes of Software Change*. London: Academic Press. 1985.

[2] Warren, I. (ed.) *The Renaissance of Legacy Systems*. Practitioner series, Springer. 2000.

[3] Bennet, K. *Legacy Systems: Coping with Success*. IEEE Software, 12(1). 1995.

[4] Tilley, S. *Perspectives on Legacy Systems Reengineering*. Reengineering Centre, Software Engineering Institute (SEI), Carnegie Mellon University, USA. 1995.

[5] Ulrich, W. M. *The Evolutionary Growth of Software Reengineering and the Decade Ahead*. American Programmer, 3(10). 1990.

[6] *Reengineering Technology Report.* Software Technology Support Centre (STSC). 1993.

[7] *The Renaissance Method*. RENAISSANCE project deliverable, D4.2. 1998.

[8] *Client/Server Migration*. RENAISSANCE project deliverable, D3.1. 1998.

[9] Ransom, J., Sommerville, I., and Warren, I. *A Method for Assessing Legacy Systems for Evolution*. Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR). 1998.

[10] *Architectural Modelling*. RENAISSANCE project deliverable, D3.2. 1998.

[11] *Evolution Planning*. RENAISSANCE project deliverable, D3.3. 1998.

[12] Sneed, H. M. *Planning the Reengineering of Legacy systems*. IEEE Software 12(1). 1995.

[13] Imai, M. Kaizen: *The Key to Japan's Competitive Success*. McGraw-Hill Publishing. 1986.

6. Acknowledgements

This work has been partially funded by the European commission's ESPRIT initiative (project 22010, RENAISSANCE). Copies of project deliverables, which describe this work in more detail and other aspects of the RENAISSANCE project, may be downloaded from http://www.comp.lancs.ac.uk/projects/renaissance.