

# Automated Software Renovation.

## 1. Introduction

The key challenge for future business operations is *change*:

1. Business requirements change whenever there is a chance of higher profitability.
2. At the same time, information and communication technologies that can be used to support business processes are innovated continuously.

These two forms of change cannot be seen in isolation: many new ways of doing business can only be realized by the use of new technologies, while innovation in software technology can lead to the emergence of completely new markets. The typical example is e-commerce: initiated by the technological developments of the world-wide web, it has now resulted in ways of doing business unthinkable ten years ago.

The flexible organization will have to align changes in business and technology. The rapidly growing integration of supply chains requires integration of systems both inside and between organizations. Software systems from the past ("legacy" systems) have never been built with that objective. The emergence of third-party software for EDI Clearing Houses, E-procurement (buying/selling via the Internet), auctions, and shopbots requires that legacy systems are used for completely new purposes using new, different, distribution channels like interactive television, WAP, AutoPC and others. There is a major gap between Internet-based technologies like XML and Java that are being used in new applications and the technologies used in legacy systems.

There is a fundamental bottleneck to alignment: legacy systems tend to be very hard to adapt. The lack of knowledge of the old systems and the need to train new users of these systems (when used in renovated form) make the potential adaptation and prolonged usage of legacy systems difficult. Again, e-commerce is a convincing example: web-enabling, for example, retail banking services requires breaking up a bank's back office system. Such systems are generally old, written in Cobol, and difficult to adapt. Consequently, they cannot keep up with the changes required from the business, potentially leading to loss of market share for the organization involved.

**Software renovation prepares a legacy system for future change.** It is based on the view that an organization's software systems provide valuable functionality, that has been proven in practice. As such, it should be reused whenever possible. At the same time, the packaging of this business functionality is usually far from optimal as they are often based on old languages, database systems, and transaction monitors, monolithic in design, and unmaintainable as a result of repeated undocumented modifications. As a consequence, legacy systems are very hard to change (if not immutable) and prohibit the alignment we aim at.

In this paper, we take a closer look at software renovation and explain how legacy systems can be renovated in three steps:

- Find Components,
- Global Restructuring,
- and Renovate per Component.



The techniques used to achieve this are analysis, transformation, and regeneration, covered in Sections 4, 5 and 6. In the current section, we provide figures on the size of the legacy problem, and summarize the historic roots of the area of software renovation.

### 1.1. The size of the legacy problem

It is tempting to assume that we might be able to deal with legacy systems by just throwing them away. However, the sheer volume of legacy software running world-wide prohibits such an approach. If we take a look at some indicators provided by [4], we see that the total volume of all software world-wide is  $7 * 10^9$  function points. The majority of software is written in old, inflexible languages. For example, 30% is written in COBOL, which corresponds to  $2.2 * 10^{11}$  statements. For mainframe applications, 80% of the software is written in COBOL.

Table 1: Forecast for numbers of programmers (worldwide) and diistribution of their activities. *source [4]*.

Year	New Projects	Enhancements	Repairs	Total	%New	%Maint
1950	90	3	7	100	90	10
1960	8,500	500	1,000	10,000	85	15
1970	65,000	15,000	20,000	100,000	65	35
1980	1,200,000	600,000	200,000	2,000,000	60	40
1990	3,000,000	3,000,000	1,000,000	7,000,000	43	57
<b>2000</b>	<b>4,000,000</b>	<b>4,500,000</b>	<b>1,500,000</b>	<b>10,000,000</b>	<b>40</b>	<b>60</b>
2010	5,000,000	7,000,000	2,000,000	14,000,000	36	64
2020	7,000,000	11,000,000	3,000,000	21,000,000	33	67

Moreover, when an industry approaches 50 years of age--as is the case with computer science-- it takes more workers to perform maintenance than to build new products. Based on current data, Table 1 shows extrapolations for the number of programmers working on new projects, enhancements and repairs. In the current decade, six out of ten millions programmers are working on enhancement and repair projects. The forecasts predict that by 2020 only one third of all programmers will be working on projects involving the construction of new software. In a world that evolves at Internet time, it is dangerous to make any extrapolations. It is unclear how developments like, for instance, application service providers and outsourcing will affect the ratio between new development and maintenance. Nonetheless, we feel that these figures are at least useful to understand the historic evolution until the current date.

These figures show that maintenance and renovation of software that exists today, is an activity of major economic importance. Since the total amount of software will only grow, the importance of maintenance and renovation will grow accordingly.

These figures also indicate that automation of this labour intensive industry will become more and more critical to enhance both quality and productivity.



## 1.2. What is software renovation?

**Software renovation is pro-active software maintenance.** It aims at improving the overall quality of software systems, making the functionality of these systems easier to use in and adapt to new application areas. An overview of this area can be found in [4,7].

Software renovation has its roots in a number of related areas. Since it is not at all uncommon that the only reliable information about what a legacy system does is the source code itself, **reverse engineering is an essential aspect of software renovation.** Originally, the notion of reverse engineering comes from hardware technology and denotes the process of obtaining the specification of a complex hardware system. In software reverse engineering, we try to distill as much useful information as possible from the system's legacy sources. The key challenge of reverse engineering is to arrive at non-trivial, higher levels of abstraction than just the source code.

Software renovation also includes modifying the software in order to improve it. Such transformations may remain at the same level of abstraction (for example, goto elimination, dead code, dead data elimination), in which case we speak of system **restructuring** or **refactoring** [2].

The alternative is to perform the renovation via a reverse engineering step, which is called **re-engineering**: first a specification on a higher level of abstraction is constructed, then a transformation is applied on the design level, followed by a forward engineering step based on the improved design.

Furthermore, software renovation has its roots in compiler and programming language technology [1 and [Asetechs-GT8 core competency](#)]. Building a compiler for a language and translating that language to assembly, involves significant analysis and transformation of programs - technology that can be reused when analyzing and transforming software for renovation purposes [3 and [Asetechs-GT8 core competency](#)].

Last but not least, the year 2000 problem and the Euro conversion have strongly affected the area of software renovation. They have resulted in an increased awareness of the need for tools for reverse and re-engineering, adoption of source analysis and modification tools, and they have increased the maturity of the renovation tools and techniques available.



---

*Asetechs-GT8 automates the activities of Software Renovation: Redocumentation, Restructuration, Restoration, Re-engineering.*

---

## 2. Renovation Target: Component-based Software

The aim of software renovation is to prepare a legacy system for future change.

A technique to arrive at the required level of flexibility is to decompose the system into a set of components. In this section, we take a closer look at component technology, and its role in software renovation.

### 2.1. Components

Following [9] components are binary units of independent production, acquisition, and deployment, that interact to form a functioning system. There are several reasons why software systems assembled from components are better capable of dealing with change, be it in business or in technology:

1. Components are independent, so implementation changes to one component can be made without affecting the others.
2. Components add abstraction, hiding implementation details behind explicit interfaces, making it safe to change this implementation when needed.
3. Components can be easily replaced by other components offering at least the same set of services.
4. Components support reuse, so that new systems can be quickly built by assembling them from existing pieces, leading to short lead time to market for new system (extensions).
5. Component systems can be a mixture of self-made and bought components, making it possible to benefit from new developments in commercially available components.
6. Components support customization, making it possible to produce a tailor-made component. This can be achieved by inheriting functionality from an existing class of components, or by setting attributes steering the component's behavior. As such, components provide a middle path between unadaptable standard packages and (expensive) bespoke customer solutions.

The technical organization of collections of components can vary. The most common form is the packaging of components in a class library. An application programmer can use the class library by writing a ``main" program that makes calls to the various classes. The application programmer has to understand the (possibly large and complex) structure of the library, but is in complete control of the use of the classes. Contrast this with component frameworks which provide canned applications that only have to be customized by writing some plug-ins that perform non-default behaviour.

From a renovation point of view, **an important property of components is that the interfaces also hide the age of the components, permitting cooperation (co-existence) of legacy components and newly created system parts** [10, 11]. This is needed in many situations, for example when web-enabling a back office system. The back office is a legacy system that has to be renovated in order to allow connections with the web. The web-interface itself is newly built.

From a legacy system (e.g., the back office) we extract components that represent the essential functionality of the legacy system. At the same time, new business requirements (e.g., web-enabling) lead to the construction of new components. The ultimate goal is to find component-based architectures and development methodologies that enable the seamless integration of reno-



vated and new components. This also implies that there should be a strong relationship between techniques for renovation and techniques for construction.

It is important to identify when combining old and new components can be applied with success:

1. The legacy components that are reused should be stable: only limited modifications are expected in the future.
2. The data models used by the legacy components and new components must be kept consistent.
3. The technical knowledge and tools needed for maintaining the legacy components will remain available in the organization.

## 2.2. Component Integration

Components do not operate in isolation, but interact to form a functioning system. This requires agreement between components on ways of communication, for which several standards exist, such as COM+ from Microsoft, Corba from the OMG group, and Enterprise JavaBeans (including RMI) from Sun.

It is our thesis that a rigorous separation of coordination (control) and computation (semantic) is the key to successful renovation. The inflexibility of traditional code is largely due to the fact that coordination and computation are completely entangled. This makes it hard to understand and restructure traditional code. This also follows the mantra of work-flow management systems, with “work” corresponding to the computation layer, and flow management to the coordination layer.

How can the conceptual separation of coordination and computation be mapped on standard middleware which is likely to be used in large projects but does not provide such a clear separation? Of course, the separation of coordination and computation can be used as guidance during technical design. In addition, we offer the following suggestions:

1. Introduce a separate “work” flow engine to implement coordination.
2. Use design and implementation standards that forbid complicated control patterns inside components.

## 2.3. Domain-Specific Component-Based Software

Following [6], the current excitement about component-based development results from the convergence of four phenomena originating from quite different backgrounds:

1. On the scientific side, the progress in the area of systematic software reuse;
2. On the industrial side, the widespread success of components for GUIs, databases, and so on, such as Microsoft's VBX, OCX, and ActiveX;
3. On the political side, the push by major players for standards such as Corba, COM, and Enterprise JavaBeans;



#### 4. In the software world at large, the generalization of object technology.

One of the key lessons of the first phenomenon, progress in the research in systematic software reuse, is that substantial benefits of component technology can only be achieved by concentrating on a particular application domain. As an example, Jacobson et al. [3] sketch the steps that a typical organization may go through when adopting component technology: an organization starts with informal code reuse leading to some reduction in development, and step by step gets closer to the domain-specific reuse-driven organization, which is capable of rapid custom product development.

Technologies such as CORBA, COM and JavaBeans provide integration at the “plumbing and wiring” level. Agreement at the domain-specific (semantic) level, however, is needed to fully arrive at the benefits of component technology, such as shorter lead time to market, appropriate customization facilities, and smooth integration of bought components. This may be achieved by further standardization. For example, as part of the CORBA effort, domain specifications exist for electronic commerce, finance, manufacturing, and so on. Also, data exchange standards based on XML, such as Open Financial Exchange (OFX) and XMLIFE for life insurances can facilitate integration of software components at the domain level.

However, arriving at such domain definitions, be it within one organization or as part of an international standardization effort, is a painful activity. It critically depends on

- the maturity of the domain;
- the level of understanding of the domain;
- the existence (or absence) of standardization efforts in the domain;
- the commercial players in the market; and
- the benefits that the various stakeholders will obtain.

We can define the notion of a domain as a family or set of systems including common functionality in a specified area [7].

Such a set of systems is worth examining if it is:

- mature, i.e., a set of legacy systems exists;
- reasonably stable, i.e., at least some of the legacy systems are worth examining;
- and economically viable, i.e., new systems are anticipated in the domain.

In our search for components, the legacy artifacts act as:

- an empirical basis for systematic scoping of domain definitions;
- a source of domain knowledge;
- potential resources to use in reengineering.

**The ultimate goal of software renovation is to make this knowledge, embodied in legacy systems, explicit, intelligible and shareable.**



### 3. Overall approach to Software Renovation

**The aim of software renovation is to understand, transform and regenerate a legacy system in such a way that its alignment with new business objectives and new technological developments is facilitated.**

We want to achieve this by separating coordination from computation and by supporting the actual computations at the domain level. We now discuss the technical and the methodological aspects of renovation.

#### 3.1. Technical approaches

We distinguish three technical approaches to software renovation:

1. **Analysis of legacy sources:** the goal is to inspect the sources of the legacy system and extract information from them that reveals their structure, purpose and architecture (Section 4). Analysis is non-intrusive since the legacy sources are only inspected and not modified.

---

*Asetechs-GT8 automates this step thanks to its Legacy Environment Analyzers.*

2. **Transformation of legacy sources:** the goal is to systematically restructure and improve the sources of the legacy system (Section 5). Transformation is intrusive since the legacy sources are modified.

---

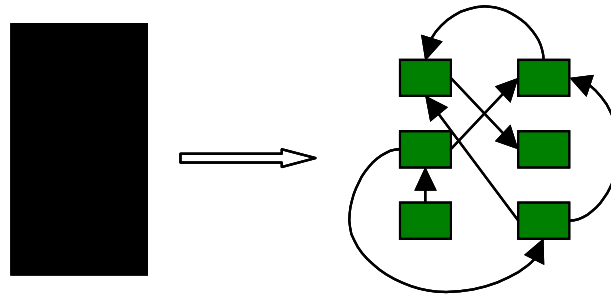
*Asetechs-GT8 automates this step thanks to its Transformations Automata. One of the most remarkable are the Relocator/Externalizer allowing to split automatically Cobol Legacy programs in 2 parts, the technical accesses to services on one side and the business functions on the other side.*

3. **Generation:** the goal is to identify potential reusable assets in the legacy system by explicitly introducing and structuring domain knowledge in such a way that major parts of the legacy system can be regenerated. Generation is also intrusive since (parts of) the legacy sources are replaced by generated code. Co-existence between legacy and renovated system is key.





If a subdivision in components is feasible continue with step B. Otherwise try other approaches. Possibilities: gradually improving the legacy system by program transformations, or replacing it by a new system.



**Step A : Subdivision of the Legacy Black-Box into Global Components**

### 3.2.2 Step B: Global Restructuring

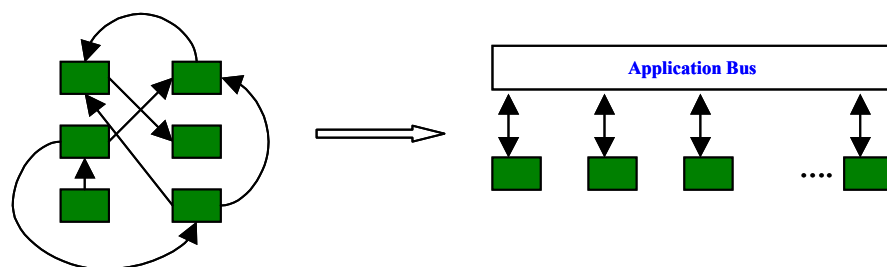
Interconnection of the components found in step A using a coordination architecture: replace the internal structure of the legacy system by a communication structure based on coordination. Legacy components (possibly wrapped to interact with the coordination architecture), appear as components. This step is only modestly intrusive since the only modifications made to the legacy are: (a) subdivision in components; (b) wrapping of the components. Major parts of the legacy remain untouched. The issues are:

- a. Reengineer the Data Bases
- b. Build a software access layer to access the Data Bases (Legacy and New)
- c. Restore the programs so that they can access the new Data Bases
- d. Externalize a wrapper for each component.
- e. Write coordination scripts to inter-connect the components.
- f. Test the restructured system.

---

*Note that points b) and c) are gradually executed and do not negatively impact on the the normal functionality of the system (Asetechs-GT8 can automate these transformations). Following point c) the database will be completely renewed, therefore all the new programs written can use the access layer to work with the renewed data base. Following to the renewal of the data base, this phase will be sensitively simplified in that the legacy systems are data oriented.*

---



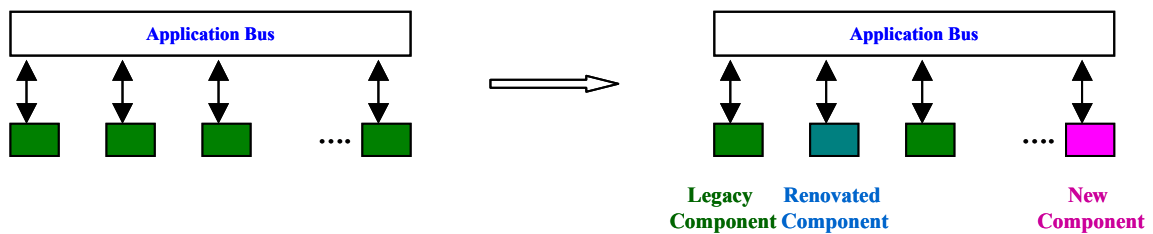
**Step B : Global Restructuring using Coordination Architecture**



### 3.2.3 Step C: Renovate per Component

Renovation of individual components. Now is the time to consider the renovation or replacement of individual components as sketched in Figure (Step C). The issues are:

- a. Determine for each component which renovation strategy to use. The options are:
- b. Leave as is.
- c. Completely replace it by commercial off-the-shelf software.
- d. Perform a very detailed analysis of the component in order to extract its business logic that can be used to rebuild or regenerate the component.
- e. Apply a layered approach: steps A, B and C are applied at the component level.
- f. Apply the selected renovation strategy.
- g. Test the component.



Step C : Renovation per Component



## 4. Analysis Techniques for Legacy Sources

The goal of analysis is to extract information from legacy systems that reveals their structure, purpose, and architecture.

Analysis techniques are crucial for *step A, Find Components*, and in this section we cover those analysis techniques that can help to find candidate components in a legacy system.

**The search for components in legacy systems is an interactive process.**

Tools collect all sorts of information about the system, which is used by the renovation engineer to find candidate components. There is no single way to find all good components: therefore, the tools should provide many different views on one legacy system, and show potential combinations of these views. This requires a hypertext-based browsing mechanism, potentially supported by intelligent agents (automata) producing specialized reports, as well as an on line annotation mechanism.

The interactive search for components can have any one of a number of different starting points.

- The first is to interview, if possible, the users, maintainers, and designers of the legacy system in order to get a picture of the overall functionality of the system, and, more specifically, to get an impression which functionality should be preserved and which is redundant.
- Another starting point is to use the persistent data stores. Of all the different sorts of data playing a role in the legacy system, it is likely that the data stored in the database represents business (domain-specific) entities. Following such data down to the actual computations (program/data CRUD Analysis) leads to those programs or procedures that could be candidate (domain-specific) components.
- Another starting point consists of the program call relationships (inter-programs relationships analysis), which may tell us something about cohesion and coupling of modules, or about the layers built into the legacy system. If one procedure invokes many others (high fan-out), and doesn't get invoked itself, it is likely to be a control (coordination) module, with little built-in functionality. Likewise, if a procedure is called by many others (high fan-in), it is likely to be some sort of utility routine, dealing with error handling or logging. The procedures with both low fan-in and low fan-out are the ones that are likely to contain business logic.
- Yet another starting point can be the screens/reports used in the system [8]. The screen/reports sequence can be identified, together with the key strokes leading to each subsequent screen/reports. Such screen/reports sequences are very close to use cases, telling what actions an end user performs. Moreover, following the flow of screen input fields through the program identifies those program slices that implement the given use case. This form of analysis can very well be supported by automated, interactive, tools.
- Techniques for combining legacy elements into novel ways in order to arrive at coherent components include concept analysis and cluster analysis. Such techniques can be used to spot combined usage of pieces of data and functionality. For example, they can be used to group data elements into candidate classes, based on their usage in programs or procedures -- which can then be made into methods of the derived class. In particular concept analysis can be used to display the various combination possibilities in a concise and meaningful manner.



- Last but not least, the search can be for a component displaying some specific sort of behavior, for example, a candidate component for valuing stock options. A hypertext-based legacy browsing system can provide various starting points for such a search, such as indexes on words occurring in comments, column names, inferred types, and so on. Moreover typical computations necessary for the behavior of the component looked may be identified using plan recognition - and these computations may then be packaged into the required component.

---

*Asetechs-GT8 proposes a 'Component Computations Recovery Process' with a set of Automata supporting its repetitive activities in order to contribute to the Legacy Detailed Analysis.*



## 5. Transformation Techniques for Legacy Sources

Transformation techniques perform intrusive, systematic, modifications of the legacy system in order to enable their maintenance/evolution and increase their flexibility.

### 5.1. General Techniques

With the insights gained by applying analysis techniques we want to transform the source code of a legacy system to:

- a. globally restructure the whole system;
- b. restructure the code of individual components;
- c. apply uniform comment conventions;
- d. eliminate deprecated language features;
- e. convert to a new language version;
- f. Translate to another language.

The transformations should be carried out by a Asetechs-GT8 automata for renovation. The renovation automata have usually three inputs:

- a. The sources of the legacy system.
- b. The repository resulting from the system analysis phase.
- c. A set of Automata Detailed Requirements specifications for Transformation.

---

*Note that: The transformation rules depend on the requirements and are thus project specific. The time needed to write these transformations ranges from one hour to several weeks. All inputs are fed to a rewriting engine that applies the transformations to the legacy sources. The output is renovated components.*

---

The state-of-the-art in renovation automata that Asetechs-GT8 can deliver today, can be characterized as follows:

- a. Automata are designed to work for one exhaustive system environment (example: Cobol, CICS, DB2/sql, jcl etc..).
- b. Transformations are implemented as full compilers' operations; and do guarantee the syntactic correctness of the results.
- c. The Asetechs-GT8' sophisticated generic frameworks are based on Attributed Trees driven Transformations.
- d. Typical performance: 1 Million Lines Of Code/hour (processes measured in millions of parts are one indicator of industrialization).

Although it is appealing to speculate about fully automatic renovation, we want to separate fact from fiction here.

It is a fiction to expect universal tools that can automatically renovate any system in any language. The same holds for universal tools that can extract business logic from arbitrary systems.



---

*Asetechs-GT8 prefers to promote Automation whenever it is possible, but only when it is economically pertinent. Automata must remain efficient 'tools' in a human controlled process.*

---

The crucial elements in successful renovation tools are:

1. Language knowledge (automated semantic analysis required).
2. Domain knowledge.
3. System Knowledge.
4. Requirements and strategy.

It is a fact that major parts of the renovation process can be automated by combining human insight with full automation of repetitive tasks (speed, quality, reproducibility, traceability along with human intelligence).

## 5.2. Support for Step B: Global Restructuring

Given the subdivision of the legacy system resulting from Step A, we now want to replace the direct connections between legacy components into indirect connections that are handled by the coordination architecture. Note that this step is only moderately intrusive: the system is reorganized into components but the code of each component is hardly touched.

The following steps are needed to achieve this:

- a. Reengineer the Data Bases
- b. Build a software access layer to access the Data Bases (Legacy and New)
- c. Restore the programs so that they can access the new Data Bases
- d. Identify the level of granularity at which the legacy system will be decomposed. Considerations are:
  - i. The total number of components should be manageable.
  - ii. Smaller components with many cross relationships are better handled as a single, larger, component.
  - iii. Wrap the identified components in order to connect them with the coordination architecture. In many cases this means interception of the ingoing and outgoing calls and replacing them by appropriate interaction with the coordination architecture.
  - iv. Write coordination scripts that simulate the connectivity in the original legacy system.

## 5.3. Support for Step C: Renovate per Component

Given the outcome of Step B, we can start the renovation of individual components. This is a completely intrusive process that may completely change the original code of the component. The most interesting properties of this approach are:



- a. The mutual dependencies between the renovation projects of the various components have been eliminated or minimized.
- b. The renovation strategy may differ per component: some components may be replaced by bying an existing commercial package while others, that contain business-specific knowledge, will undergo a very detailed analysis and restructuring.

In those cases that the decision is to perform a detailed renovation based on the existing code--typically when much usefull business logic is contained in it--transformation techniques may be applied to the code of the component. Typical issues are:

- a. Transformations aiming at code improvement (e.g., applying uniform layout conventions, goto elimination, code restructuring, and dead code, dead data elimination).
- b. Transformations aiming at the replacement of certain properties of the code (e.g., change of the user-interface or the database engine).
- c. Full translation of the code to another language or platform (e.g., conversion between COBOL dialects or translation from obsolete 4GL to standard 3GL).

---

*Asetechs-GT8 proposes specific 'project automata' developments in order to automate all of these activities.*



## 6. Conclusions

The major challenge for future business operations is to align changing business goals and changing technologies, while preserving the assets that are hidden in the legacy systems supporting today's business operations.

In this paper we have formulated the main questions system renovation has to solve, and we have given a comprehensive overview of techniques and approaches.

We have discussed the automated analysis and transformation of legacy systems. By stressing the need for cooperation (even co-existence) between renovated software and new software we naturally arrived at the need for component-based approaches and coordination architectures that define the cooperation between old and new components.

We have presented a three step approach to system renovation:

1. find components,
2. global restructuring,
3. and renovate per component.

The necessary techniques for analysis, transformation, and regeneration of legacy systems were also discussed.

Our main conclusions are:

1. Maintenance and Renovation are economically motivated activities.
2. Legacy systems are a valuable asset for business operations.
3. Automation is a powerful, robust, available and affordable means to industrialize Software Renovation.

Software Renovation aims at extracting these assets in the form of reusable components.

Component-based architectures enable the cooperation/co-existence between renovated and new components and enable the up-coming of Industrialized Pro-active Maintenance Processes.

The key success factors for Software Renovation are the following:

1. Renovation methods and techniques should deliver reuseable components.
2. One should use software architectures and development methods that enable the integration/co-existence of renovated and new components.
3. Software Renovation should be thought of as the future run of the mill Software Maintenance Practice of the software development departments.



## 7. Bibliography

- 1 A.V. Aho, R. Sethi, and J.D. Ullman.  
Compilers. Principles, Techniques and Tools.  
Addison-Wesley, 1986.
- 2 M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts.  
Refactoring: Improving the Design of Existing Code.  
Addison-Wesley, 1999.
- 3 I. Jacobson, M. Griss, and P. Jonsson.  
Software Reuse; Architecture, Process and Organization for Business Success.  
Addison Wesley, 1997.
- 4 C. Jones.  
Applied Software Measurement: Assuring Productivity and Quality.  
McGraw-Hill, 1991.
- 5 C. Jones.  
Applied Software Measurement.  
McGraw-Hill, 1996.
- 6 B. Meyer and C. Mingins.  
Component-base development: From buzz to spark.  
IEEE Computer, 23(7):35-37, 1999.
- 7 M. Simos.  
Organization domain modelling (ODM) guidebook version 2.0.  
Technical Report STARS-VC-A025/001/00, Synquiry Technologies, Inc, 1996.  
URL: <http://www.synquiry.com/>. 450 pp.
- 8 E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuck.  
Reverse engineering legacy interfaces: An interaction-driven approach.  
In 6th Working Conference on Reverse Engineering, WCRE'99, pages 292-301. IEEE Computer Society, 1999.
- 9 C. Szyperski.  
Component Software; Beyond Object-Oriented Programming.  
Addison-Wesley, 1998.
- 10 G. Visaggio,  
Ageing of a Data-Intensive Legacy System: Symptoms and Remedies, Journal of Software Maintenance: Research and Practice, John Wiley; vol. 13, pp. 281-308, 2001.
- 11 G. Visaggio,  
Value-based decision model for renovation processes in software maintenance, Annals of Software Engineering, 9, Kluwer Academic Publishers, May 2000, pp 215-233.

