From Legacy to Component: Software Renovation in Three Steps

Arie van Deursen
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
http://www.cwi.nl/~arie/

Ben Elsinga
Program Director CBD and Security, CAP Gemini
P.O. Box 2575, 3500 GN Utrecht
Ben.Elsinga@capgemini.nl

Paul Klint
CWI and University of Amsterdam
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Paul.Klint@cwi.nl

Ron Tolido
Corporate Technology Officer, CAP Gemini
P.O. Box 2575, 3500 GN Utrecht
Ron.Tolido@capgemini.nl

Abstract

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 formulate the main questions *system renovation* has to solve, and we give a comprehensive overview of techniques and approaches. We discuss the analysis and transformation of legacy systems and also describe how domain engineering can help to identify reusable domain knowledge. By stressing the need for cooperation between renovated software and new software we naturally arrive at the need for component-based approaches and coordination architectures that define the cooperation between old and new components. We present a three step approach to system renovation: find components, global restructuring, and renovate per component. The necessary techniques for analysis, transformation, and regeneration of legacy systems are also discussed. The paper concludes with a description of domain engineering and domain-specific languages as viable techniques for the structuring and reuse of the application domain knowledge that is embedded in legacy systems.

Acknowledgements. This research was funded by CAP Gemini. Daan Rijsenbrij (Corporate Scientific Officer, CAP Gemini) initiated the project that has resulted in this paper.

© *CAP Gemini*, 2000

Contents

1 Introduction **3**
- 1.1 The size of the legacy problem 5
- 1.2 What is software renovation? 5

2 Renovation Target: Component-based Software **6**
- 2.1 Components 6
- 2.2 Component Integration 8
- 2.3 Domain-Specific Component-Based Software 9

3 Overall approach to Software Renovation **11**
- 3.1 Technical approaches 11
- 3.2 Methodological approach 12

4 Analysis Techniques for Legacy Sources **15**

5 Transformation Techniques for Legacy Sources **16**
- 5.1 General Techniques 16
- 5.2 Support for Step B: Global Restructuring 18
- 5.3 Support for Step C: Renovate per Component 18

6 Generation Techniques for Legacy Sources **19**
- 6.1 Domain Engineering 19
- 6.2 Domain-specific Languages 20
- 6.3 Risla: a DSL in the Financial Domain 21

7 Conclusions **21**

1 Introduction

The key challenge for future business operations is *change*: Business requirements change whenever there is a chance of higher profitability. 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 five 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,

Year	New projects	Enhancements	Repairs	Total	% New	% Maint.
1950	90	3	7	100	90	10
1960	8500	500	1000	10000	85	15
1970	65000	15000	20000	100000	65	35
1980	1200000	600000	200000	2000000	60	40
1990	**3000000**	**3000000**	**1000000**	**7000000**	**43**	**57**
2000	4000000	4500000	1500000	10000000	40	60
2010	5000000	7000000	2000000	14000000	36	64
2020	7000000	11000000	3000000	21000000	33	67

Table 1: Forecasts for numbers of programmers (worldwide) and distribution of their activities

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 prolongated 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. In Section 2 we argue that the goal of renovation is to arrive at *component-based* software systems, which permit the smooth integration of newly built software and renovated legacy components. In Section 3 we then explain how legacy systems can be componentized 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 [15], 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.

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 [16, page 319], Table 1 shows extrapolations for the number of programmers working on new projects, enhancements and repairs. In the current decade, four out of seven 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.

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), in which case we speak of system *restructuring* or *refactoring* [13]. 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]. 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].

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.

2 Renovation Target: Component-based Software

The aim of software renovation is to prepare a legacy system for future change. An essential 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 [20] 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:

- Components are independent, so implementation changes to one component can be made without affecting the others.
- Components add *abstraction*, hiding implementation details behind explicit interfaces, making it safe to change this implementation when needed.
- Components can be easily replaced by other components offering at least the same set of services.
- 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).
- Component systems can be a mixture of self-made and bought components, making it possible to benefit from new developments in commercially available components.
- 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 unadapatable standard packages and (expensive) bespoke customer solutions.

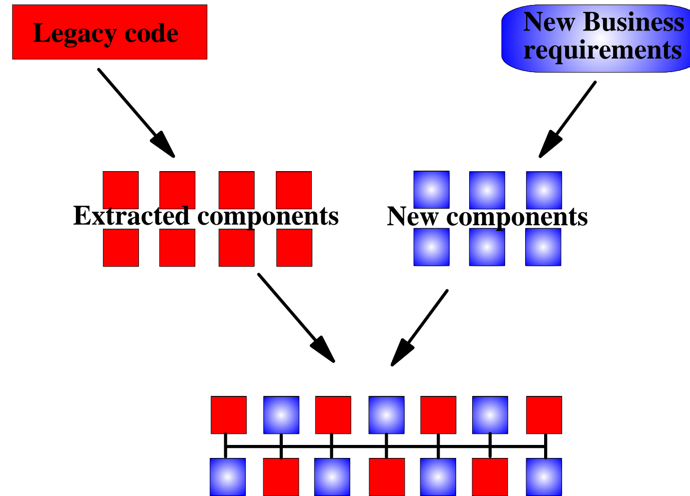


Figure 1: Integrating renovated and new components.

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 of legacy components and newly created system parts. 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.

In Figure 1, we sketch this situation. 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 renovated 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:

- The legacy components that are reused should be *stable*: only limited modifications are expected in the future.
- The data models used by the legacy components and new components can be

kept consistent.

- The technical knowledge and tools needed for maintaining the legacy components will remain available in the organization.

When one or more of these conditions is not satisfied, the only alternative is to reimplement the legacy component using new technology.

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.

For the purpose of this paper, we use a *software bus* approach to illustrate the concepts of component integration, as this clearly emphasizes several aspects of component technology, such as independence, binary deployment, distribution, and heterogeneous implementations. In large applications more or less standard technology like, for instance, COM+ or Corba, will be used as middleware layer. These technologies have, however, a steep learning curve and are conceptually and technically quite intricate.

For expository purposes we prefer therefore the use of a much simpler software bus called the ToolBus, which has been developed at CWI and the University of Amsterdam [2]. It is a programmable bus with the following characteristics:

- Components (”tools” in ToolBus parlance) are connected to the ToolBus.
- The cooperation between components is described by a process-oriented script in the ToolBus.
- Unlike COM+, Corba and others, the protocol of cooperation is made explicit.
- Tools can be implemented in different languages and can run on different machines.
- There is no direct communication between tools.

The ToolBus architecture is sketched in Figure 2. At the bottom we see tools (blue boxes) that have a bidirectional connection with the ToolBus. Tools capture the *computation* layer, which are the programs that carry out specialized tasks such as sorting a file, displaying a window, or computing the estimated value of futures through a bank's back office systems.

On top we see the ToolBus itself (grey rectangle). Inside the ToolBus several processes are executing (red circles). ToolBus processes can communicate with each other as well as with tools. The ToolBus processes capture the *coordination* layer, which deals with the way in which program and systems parts interact (by way of procedure calls, remote method invocations, and the like). In order to enable the exchange of data, a common data format is used comparable to XML.

It is our thesis that a rigorous separation of *coordination* and *computation* is the key to successful renovation. The inflexibility of traditional code is largely due to the

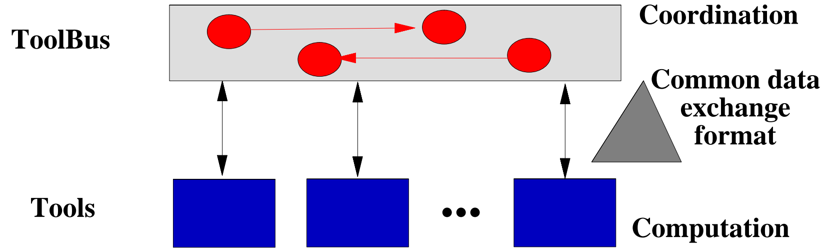


Figure 2: The ToolBus Coordination architecture.

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:

- Introduce a separate workflow engine to implement coordination.
- Use design and implementation standards that forbid complicated control patterns inside components.
- Use a widely accepted standard like XML for data exchange.

2.3 Domain-Specific Component-Based Software

Following [17], the current excitement about component-based development results from the convergence of four phenomena originating from quite different backgrounds:

- On the scientific side, the progress in the area of systematic software reuse;
- On the industrial side, the widespread success of components for GUIs, databases, and so on, such as Microsoft’s VBX, OCX, and ActiveX;
- On the political side, the push by major players for standards such as Corba, COM, and Enterprise JavaBeans;
- 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* [18, 21]. As an example, Jacobson *et al.* [14] 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, manifacturing, 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* [18]. Such a set of systems is worth examinig 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.

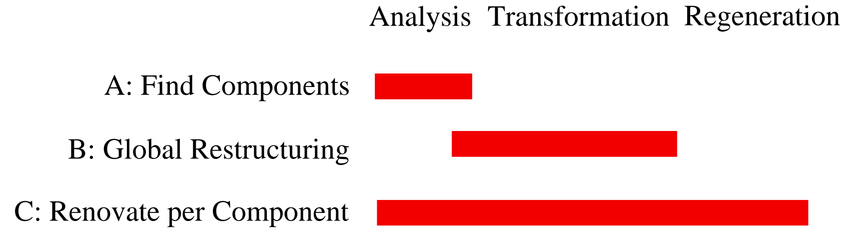


Figure 3: Relation between technical and methodological aspects of renovation

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. The relation between the two is sketched in figure 3: each methodological step uses certain technological approaches.

3.1 Technical approaches

We distinguish three technical aproaches to software renovation:

- *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.

- *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.

- *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 (Section 6). Generation is also intrusive since (parts of) the legacy sources are replaced by generated code.

Analysis and transformation of legacy systems are closely related as is shown in Figure 4: analysis gathers information that can be used, amongst others, for transformation. Generation based on domain engineering is a higher level approach that *uses* information from the analysis phase and can exploit transformation techniques to achieve its goals.

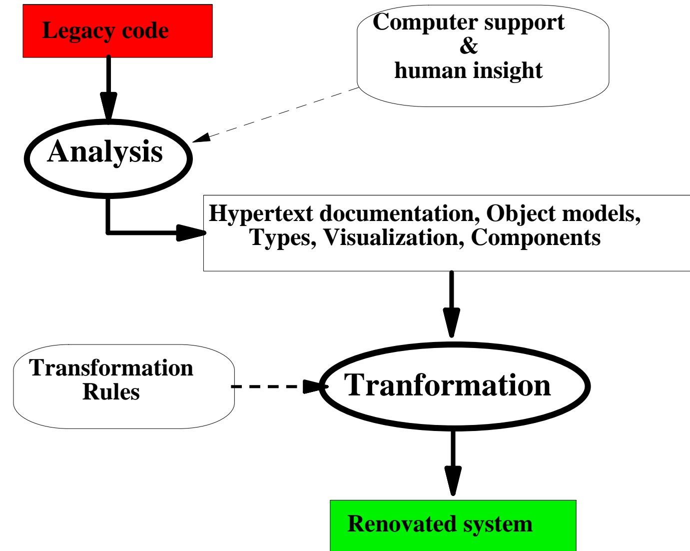


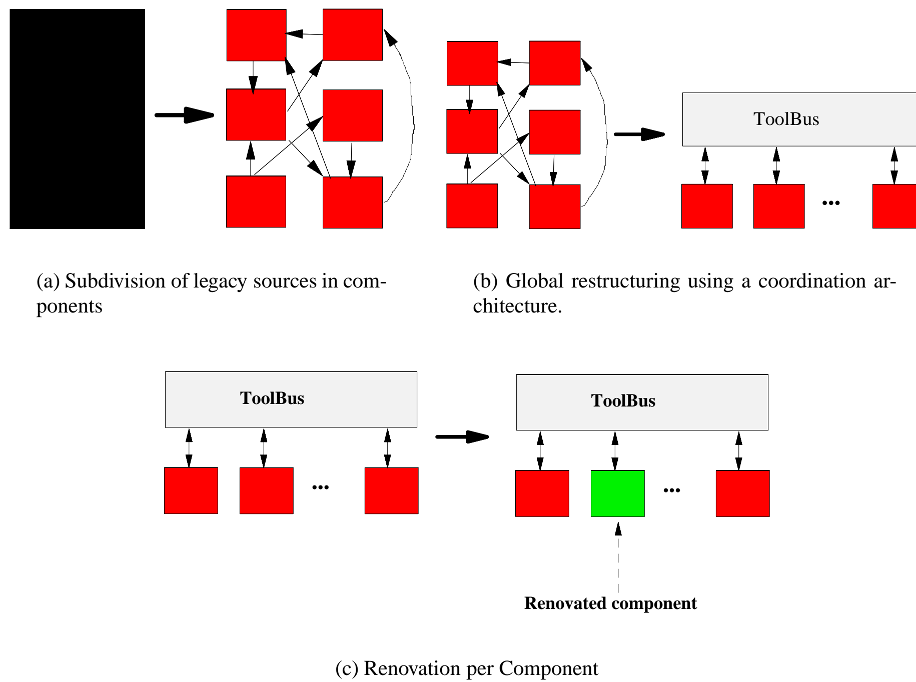
Figure 4: Analysis and transformation of legacy systems

3.2 Methodological approach

Given the various technical approaches to renovation (analysis, transformation, generation), how can we manage the software renovation process? We describe a three step approach:

Step A: Find Components Subdivision of the legacy sources in components: given a legacy system as black box, discover the internal structure of this box. This is illustrated in Figure 5(a). This step is *non-intrusive* and the following issues play a role here:

- Are all sources of the legacy available?
- For missing sources, use tools to recover the source from binaries.
- Determine the correspondence between source code versions and versions of binaries.
- Determine the languages used in the legacy system.
- Obtain analysis tools for these languages, for instance, by instantiating a generic tool set for them.
- Run the analysis tools on the legacy system and interpret the results.
- Determine whether a subdivision in components is feasible and where the component boundaries should be.



(a) Subdivision of legacy sources in components

(b) Global restructuring using a coordination architecture.

(c) Renovation per Component

Figure 5: Renovation in Three Steps

- 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 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:

- Write a wrapper for each component.
- Write coordination scripts to connect the components.
- Test the restructured system.

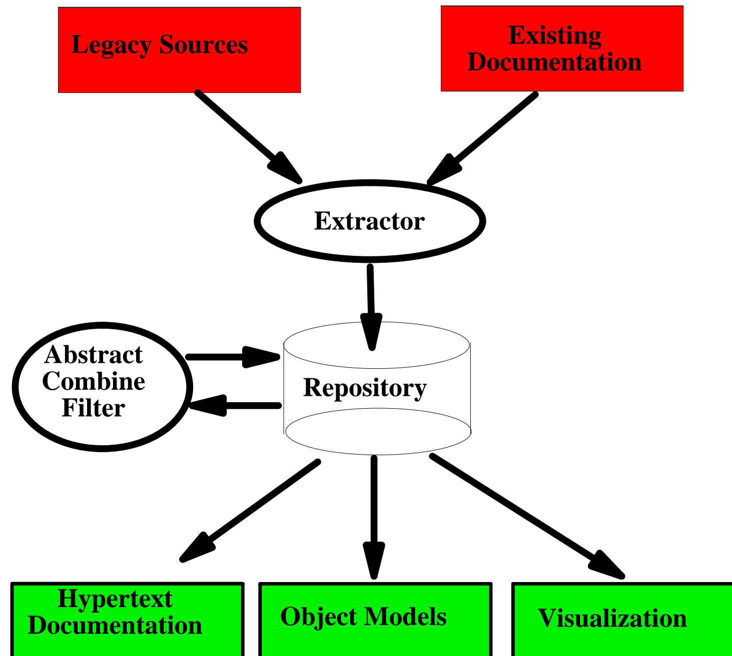


Figure 6: Architecture of a Redocumentation Tool Suite.

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 5(c). The issues are:

- Determine for each component which renovation strategy to use. The options are:
  - Leave as is.
  - Completely replace it by commercial off-the-shelve software.
  - 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.
  - Apply a layered approach: steps A, B and C are applied at the component level.
- Apply the selected renovation strategy.
- Test the 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 an intelligent agent, as well as an on line annotation mechanism. An architecture for such a program understanding tool suite is displayed in Figure 6 [9].

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 leads to those programs or procedures that could be candidate (domain-specific) components.

Another starting point consists of the program call relationships, 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 [8].

Yet another starting point can be the screens used in the system [19]. The screen sequence can be identified, together with the key strokes leading to each subsequent sreen. Such screen 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* [10]. 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 valueing 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 [11], and so on. Moreover typical computations necessary for the behavior of the component looked may be identified using *plan recognition* [12] – and these computations may then be packaged into the required component.

5 Transformation Techniques for Legacy Sources

Tranformation techniques perform intrusive, systematic, modifications of the legacy system in order to enable their maintenance 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:

- globaly restructure the whole system;
- restructure the code of individual components;
- apply uniform comment conventions;
- eliminate deprecated language features;
- convert to a new language version;
- Translate to another language.

The transformations should be carried out by a fully automated renovation factory [5, 6] as shown in Figure 7. The renovation factory has three inputs:

- The sources of the legacy system.
- (Optionally) the repository that results from the analysis phase.
- A set of problem-specific transformation rules

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.
- The transformations can be organized as a *pipeline* of elementary transformations; this promotes reuse.

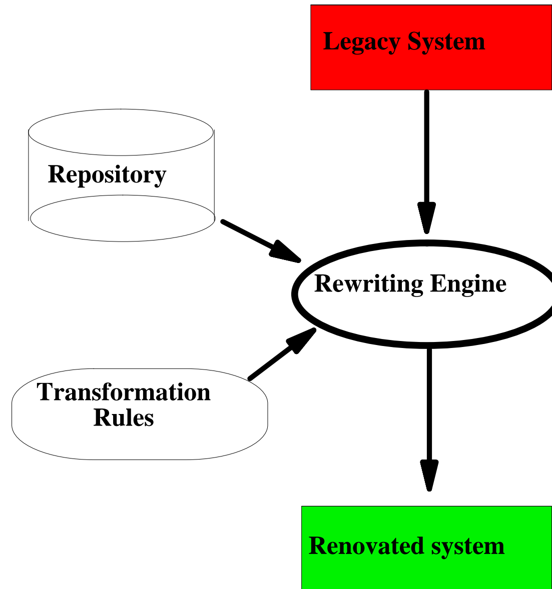


Figure 7: Architecture of a Renovation Factory.

All inputs are fed to a rewriting engine that applies the transformations to the legacy sources. The output is a renovated system.

The state-of-the-art in renovation factories can be characterized as follows:

- Most available tools work only for one language.
- Transformations are implemeneted as simple string editing operations; this does not guarantee the syntactic correctness of the result.
- The most sophisticated generic frameworks are based on *tree transformations*.
- Typical performance: 100-1000 KLOC/hour.

Although it is appealing to speculate about fully automatic renovation factories, 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.

The crucial elements in successfull renovation tools are:

- Language knowledge.
- Domain knowledge.
- System Knowledge.

- Requirements and ICT 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).

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 moderatley intrusive: the system is reorganized into components but the code of each component is hardly touched. Figure 5(b) sketches this idea.

The following steps are needed to achieve this:

- Identify the level of granularity at which the legacy system will be decomposed. Considerations are:
  - The total number of components should be manageable.
  - Smaller components with many cross relationships are better handled as a single, larger, component.
- 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.
- 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:

- The mutual dependencies between the renovation projects of the various components have been eliminated.
- 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:

- Tranformations aiming at code improvement (e.g., applying uniform layout conventions, goto elimination, code restructuring, and dead code elimination).

- Transformations aiming at the replacement of certain properties of the code (e.g., change of the user-interface or the database engine).
- Full translation of the code to another language or platform (e.g., conversion between COBOL dialects or translation from obsolete 4GL to standard 3GL).

6 Generation Techniques for Legacy Sources

The most sophisticated technical approach to renovation is to decompose the legacy system into components in such a way that these components become reusable across different applications. Customized versions of these components can then be used in different configurations.

In order to achieve these goals a deeper understanding of the application domain is needed. *Domain engineering* attempts to distill domain knowledge out of legacy systems.

First, the legacy system has to be reorganized to provide the domain knowledge at the proper level of abstraction. Second, a notation tailored towards the domain—a *Domain-Specific Language* (DSL)—has to be provided to enable the easy composition of components into a workable system. Finally, the DSL will be used as input for a generator (the DSL compiler).

The major advantages of this approach are a very short time to market achieved by extensive reuse and very effective component composition.

Examples of DSLs are EXPRESS—the information modeling language specified in STEP (ISO 10303-11)—that is used for product data representation and exchange and Risla 6.3 that is used for the description of interest-based financial product. Many other DSLs exist in areas like plant control, web-site generation, configuration management, etc.

6.1 Domain Engineering

Domain engineering is concerned with the identification and demarcation of application domains. A domain is here understood as a family of systems in a well-defined application area like finance or process control.

As a rule of fist, domain engineering pays off when there are already three legacy systems available in the given domain and when it is expected that at least three more applications in the domain will be build in the future. Domain engineering is a form of organizational learning: the experience and knowledge that is implicit in a number of legacy systems is distilled into a concise DSL and supporting generator.

The following methodological issues play a role in Domain Engineering:

- Identification of the stakeholders.
- Determination of the boundaries of the domain.
- Identification of the domain experts.

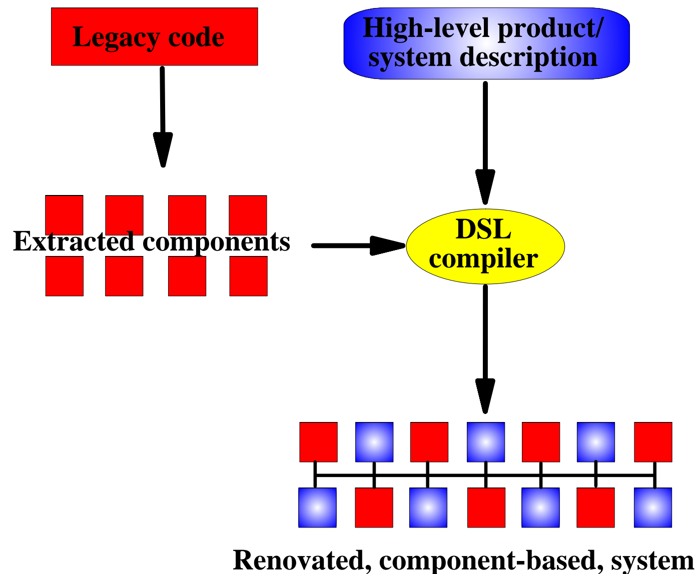


Figure 8: A domain-oriented approach.

- Reengineering of the legacy systems. Results are insights in domain concepts as well as reusable components to be used by the DSL compiler.
- Determination of a lexicon of domain-specific terms.
- Determination of the commonality and variability between the various legacy systems.
- Design of a DSL.
- Design and implementation of a generator (compiler) for the DSL.

6.2 Domain-specific Languages

The link between renovation and construction can be improved by introducing a DSL. The idea is illustrated in Figure 8 and works as follows. Instead of implementing new business requirements directly (as was done in Figure 1), we introduce high-level product/system descriptions at a much higher level. The descriptions are written in a newly designed DSL that is tailored towards the domain in question. In addition, renovated components that have been extracted from the legacy system act as a library for the DSL. From these descriptions and the component library, a DSL compiler generates a new application.

6.3 Risla: a DSL in the Financial Domain

We have already applied the approach sketched here in the financial domain. The Risla language (part of Cap Gemini's FPS product suite) is in use at, for instance, MeesPierson and ING to describe financial products. It was born out of two observations: the time-to-market for innovative products becomes shorter and a bank's backoffice is hard to adapt.

Based on domain notions (e.g., loan, swap, future, and FRA) a simple specification language has been designed that can be translated to COBOL code. A library of standard components is linked with the generated code.

Key to the success of Risla (and to future projects) are:

- Extensive domain knowledge that was embodied in a high-quality procedure library. This library captures the knowledge of a series of preceeding projects in the same domain.
- The insight that the plain use of the library itself resulted in lengthy, repetitive, unmaintainable code. By introducing a DSL and a generator that produces the calls to the library (and other additional glue code) common knowledge about the domain and the underlying infrastructure is concentrated in the generator (as opposed to being repeated in each program).

7 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 analysis and transformation of legacy systems and have also described how domain engineering can help to identify reusable domain knowledge. By stressing the need for cooperation 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: find components, global restructuring, and renovate per component. The necessary techniques for analysis, transformation, and regeneration of legacy systems were also discussed. We concluded with a description of domain engineering and domain-specific languages as viable techniques for the structuring and reuse of the application domain knowledge that is embedded in legacy systems.

Our main conclusions are:

- Maintenance and renovation are economically motivated activities.
- Legacy systems are a valuable asset for business operations.
- Software renovation aims at extracting these assets in the form reusable components. Domain-specific languages capture essential domain knowledge and can be used to (re)generate software systems.

- Component-based architectures enable the cooperation between renovated and new components.

Based on the analysis in this paper, we conclude that the key success factors for software renovation are the following:

- Renovation methods and techniques should deliver reuseable components.
- One should use software architectures and development methods that are component-based and enable the integration of renovated and new components.
- Domain knowledge can be captured in reusable components that can be exploited in combination with DSLs.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] J.A. Bergstra and P. Klint. The discrete time ToolBus—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[3] M.G.J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[4] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM Software Engineering Notes*, 22(1):57–68, 1997.

[5] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. IEEE Computer Society, 1997.

[6] J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proc. Third European Conference on Software Maintenance and Reengineering*, pages 83–91. IEEE Computer Society, 1999.

[7] A. van Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 1999.

[8] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998.

[9] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[10] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[11] A. van Deursen and L. Moonen. Understanding COBOL systems using types. In *Proceedings 7th Int. Workshop on Program Comprehension, IWPC'99*, pages 74–83. IEEE Computer Society, 1999.

[12] A. van Deursen, A. Quilici, and S. Woods. Program plan recognition for year 2000 tools. *Science of Computer Programming*, 1999.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse; Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.

[15] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 1991.

[16] C. Jones. *Applied Software Measurement*. McGraw-Hill, second edition, 1996.

[17] B. Meyer and C. Mingins. Component-base development: From buzz to spark. *IEEE Computer*, 23(7):35–37, 1999.

[18] 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.

[19] 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.

[20] C. Szyperski. *Component Software; Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[21] J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, 1996.