As business and technology evolve and software becomes more complex, researchers and tool vendors in reengineering are constantly challenged to come up with new techniques, methods and solutions to provide effective support for the transition of legacy systems to modern architectural and technological paradigms. Such pressure has been witnessed repeatedly over the past decades. Examples include the adoption of object-oriented programming languages and, more recently, the advent of web technologies, in particular service-oriented architectures (SOAs). Professor Reiko Heckel, University of Leicester, explains why software reengineering and modernisation is a job for specialists, supported by clever software tools.

As every IT professional knows, software doesn’t just sit inside the computer: it has a life of its own and evolves over time. This is what happens, for example, when we are asked if we want to download an updated version of the application that we have just launched. The fact that software needs to be continually adapted in order to deliver the same level of satisfaction to the user (or even increase it), is referred to as Lehman’s first law of software evolution.\(^1\)

Large organisations such as banks use very complex software applications, and evolving them is a much more challenging task. If one is not careful, complexity increases as software is evolved. This is known as Lehman’s second law of software evolution. Very often, layers of software keep being added without restructuring what was there already or new applications are coarsely stitched to old ones without taking into account the global structure or architecture of the system. In modern terms, we could say that software becomes ‘obese’: for example, programmers are often afraid to delete code even if it is no longer executed and code can get replicated all over the application in marginally different versions.

As it lets ‘fat’ accumulate, software becomes less and less efficient, more and more difficult to change and lacking the levels of agility, flexibility and responsiveness that companies require to address the fierce competition and market volatility that characterises business today. As with humans, one can perform a surgical operation to remove the fat. In software engineering, this requires a careful analysis of the code, breaking it into meaningful chunks so that one can understand what is ‘fat’ and what is ‘muscle’ and reorganising what is left so that the original functionality is preserved. One of the methods that has been gaining popularity for keeping software fit and agile is the adoption of an ‘architecture’, i.e. a way of organising software as a structure of interconnected (smaller) components and restricting changes on software to reconfigurations that conform to or preserve that structure. In other words, evolution has to adhere to a (strict) ‘regimen’ that allows organisations to plan and optimise the use of resources as well as control the quality of their systems.

Re-engineering and modernisation through transformations
The main techniques that are usually employed for software reengineering, maintenance and modernisation are based on program transformations. An example is source-to-source transformation, which moved from a successful research idea (e.g. TXL or ASF+SDF systems) to successful industrial tools (e.g. DMS or Forms2Net).

The problems of source-to-source transformation increase in complexity when we move up the abstraction ladder...
The problem of re-engineering at the architectural level in general, and the development of a methodology and tools for transformations from legacy systems to SOAs in particular, are still the focus of intensive research. A promising solution can be derived from the Horseshoe Model of SEI-CMU by considering the following steps (see the diagram above).

1. Code annotation, identifying for source code fragments the elements of the target architecture they will have to be assigned to;
2. Reverse engineering, producing an abstraction of the annotated source code;
3. Redesign transformation of the source into the target graph model, to obtain an abstraction of the annotated target code;
4. Forward engineering, to generate the annotated target code based on the annotated source code and the target graph model.

The redesign is achieved by graph transformations implementing model-level refactoring operations controlled by the code annotations. Rather than text – or tree-based code transformations, graph transformations allow for improved scalability due to a concise and direct representation of the architecture. The intended result is expressed by a set of constraints over the metamodel, to be satisfied when the transformation is complete.

Model-level transformations can be mapped to code-level refactoring primitives. The target code is therefore obtained from the annotated source code by applying the code-level refactorings corresponding to the model-level transformations. The result of this step has the same structure as the input to step 1, thus allowing for several iterations of the cycle, such as those addressing the technological (horizontal) and functional (vertical) dimensions of SOAs.

About the author
Professor Reiko Heckel holds a chair in software engineering at the Department of Computer Science, University of Leicester (www.cs.le.ac.uk).

Footnote
1. Manny Lehman sadly passed away in December 2010. For his early studies of the software evolution phenomenon he was known as the Father of Software Evolution.