Rewrite

Patterns for the replacement and maintenance of Software Systems.

Thomas Mey
Münchener Rückversicherungsgesellschaft
Königinstr.107
80802 München
ThomasMey@yahoo.de
tmey@munichre.com
June 2006

This paper presents a small set of patterns covering the replacement and maintenance of software systems. The patterns APPLICATION LIFECYCLE and TECHNOLOGY ROADMAP support you to choose the most appropriate point in time to replace a system. 1:1 REPLACEMENT is a strategy guiding the replacement of a system by replacing just its original functionality.

APPLICATION LIFECYCLE and TECHNOLOGY ROADMAP

Both patterns support you in deciding when to replace an application or parts of it.

The context for both patterns is identical: A productive application is in place since a considerable amount of time and new requirements demand changes in the application, but you also think about redoing the whole application.

APPLICATION LIFECYCLE: Determine at which stage your application is in the lifecycle to support the decision on how to proceed with the application.

Problem:

You cannot decide which option among maintaining, migrating and redeveloping an application is best.

Solution:

Determine at which point of the lifecycle the application is standing. To find this out use indicators relevant to the evolvement of the application as well as current and future requirements. Redevelop the application when it has reached the end of its lifecycle.

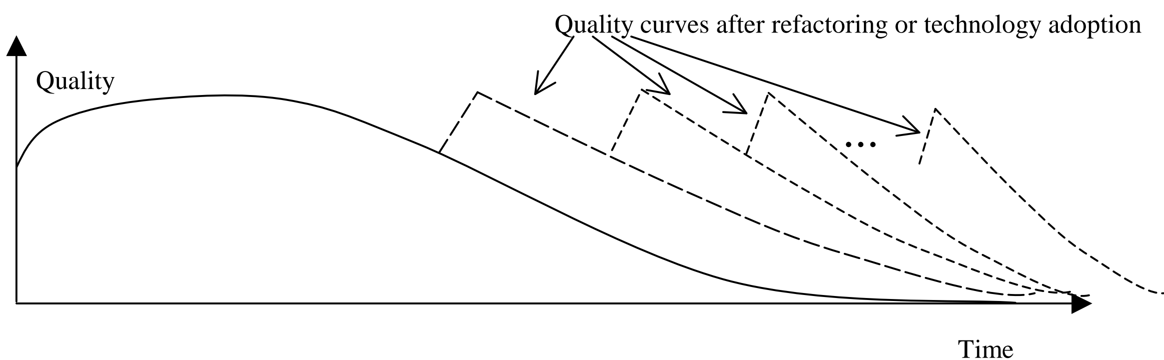
Forces:

The key assumption is that applications have a lifecycle similar to other man made artifacts like buildings or cars. All of these are built - often under time pressure - and have a more or less stable architecture when they are first released. The lifecycle can be short or long depending on the stability of the application architecture and demands of the market.

The pattern uses a retrospective and a projective approach. Lets first look into the retrospective measures that are concerned with maintenance of the software. This can help you to estimate what you have been changing and how much the change cost you.

Different forms of maintenance will be performed on Software Systems after the first release. The reasons for these changes can be classified into different categories. Use such a categorization scheme to track changes to the application. A common method to differ between changes is based

on the intention of a change, known as adaptive, corrective or perfective maintenance [Swanson]. These types of maintenance can be summarized as
Perfective: Perfect the system in terms of performance, efficiency or maintainability
Adaptive: Adapt the system to changes in its data or processing environment
Corrective: Correct failures of the system
Other classification schemes can be applied when the intention of changes is not clear. Ned Chapin with others introduce Type Clusters (Support interface, Documentation, Software properties, Business rules) for classification. This approach is easier to follow, since a decision tree has to be followed to determine the type of change [Chapin].
The reasons for change can be driven by market conditions as market volatility, customer expectations and technology. For commercial software Sahin and Zahedi introduce an other classification scheme with the following categories [Sahin]:
Warranty: Changes the producer makes in order to make sure that it works as claimed
Maintenance: Improve or enhance exiting functions
Upgrade: Add new functions or features
The categorization of changes made to system will help you in two ways: You can measure how much you are spending for changes of each kind thus being able to estimate the cost of future changes. The trade-off between effort and benefit of a change lets you estimate its age.
No matter what type of change is made to a system, for which reason and how well the system was engineered, changes to a system tend to degrade its quality, hence the system ages. A legacy system – defined as working software with documentation describing its structure and behavior – is said to be aged when it has poor quality [Visaggio, p 281-282]. Note that this definition does not take into account the actual age of the system but only refers to its quality.
The following graph shows change of quality of a system over time. While bug fixes after the initial release raise the quality in early stages a degradation process leads to quality reduction in longer terms. This reduction can be mitigated by changes to the system that raise the system quality, e.g. refactoring or adoption to new technologies.



Other methods for measuring change to a system are available, for example based on the fault enumeration [Nikora] or on symptoms many of which require reading the existing source code [Visaggio].

Let us now look to prospective measures. Some of these are the current and future requirements of the system in question. It will not or only hardly be possible to implement some of these requirements in the exiting system. Such requirements include

- The need to integrate new technologies or the end of support for old technologies
- High cost for old hardware or hardware that is no more available
- Architectural limits in the current application

Other factors influence the ability to maintain a system in the future, for example important developers who leave or retire.

Which options do you have generally in the long term evolution of a system?
When classifying a system as legacy system three options are available for the evolution of such a system: Wrapping, Redevelopment and Migration. Wrapping is the alternative of choice when no platform change is anticipated, Migration and Redevelopment are the alternatives for a platform change [Rhagozar 325-329]. Platform in the referenced context means mainframe and mini environments compared to UNIX or windows environments as target for a redevelopment or migration. The objective of APPLICATION LIFECYCLE pattern is to determine the right moment for redevelopment or eventually migration.

Resulting Context
With the knowledge of the age and the lifetime of a system investments in the maintenance of an existing system or the founding of a redevelopment can be done knowingly.
Known Uses
In Munich Re the decision to redevelop an application was based on the fact people with important knowledge of the application will retire and the technology (programming language) of the old system will no more be supported in the near future.
In a manufacturing company for Smart Card personalization systems an old system has been replaced with a new system because the old one was considered as technical obsolete, regarding the mechanical architecture and the software platform – which was C for MS DOS™.

TECHNOLOGY ROADMAP:

Establish a roadmap of technologies to constrain the lifecycle of applications.

Problem:

How do I determine when to port an application to a new technological environment or to rewrite it as a whole.

Solution:

Establish a TECHNOLOGY ROADMAP that comprises the lifecycle of technologies supported. While the date of introduction of a technology is important for new applications the date of phasing out a technology is important for existing systems. Hence both dates are important. The roadmap should also state how many different versions of one technology you will support (e.g. 'at one time only two versions of the Java runtime shall be supported'). Leaving out technologies can help you in limiting the frequency of updates (we leave out version 9 and migrate from version 8 to 10 directly).

Forces:

Technologies affecting software development change at different speed. While data store technologies and solutions change at low speed programming languages come and go more frequently. This applies predominantly to vendor specific languages as e.g. Clipper™ which has been replaced with Visual Objects™ or Turbo-Pascal™ which has been succeeded with Delphi. Operating systems regularly introduce new features and discontinue others with some regularity so that applications usually have to be adapted to new Operating System versions. Why - at first glance - should you change a running system? Commercial support for products in use is a requirement in many companies. When not adapting stepwise to a changing environment not only applications will be difficult to migrate to new technologies but also the skills of the developers who implement the systems. Applications that have been written for a specific environment may have to be adapted to a new environment, e.g. from .NET Framework Version 1.1 to Version 2.0. When an Operating System Platform changes as it was the case from Windows 3.1 to Windows NT, much adaptive work may be necessary, or when the support for a product is discontinued – as e.g. for Visual Basic 6™ (VB6) – a migration or redevelopment is necessary. At this point the APPLICATION LIFECYCLE pattern may help you to decide which direction to take.

The technologies that should be covered in the roadmap are the strategic technologies you need to run your business. For Business IT-Systems these typically include the following:

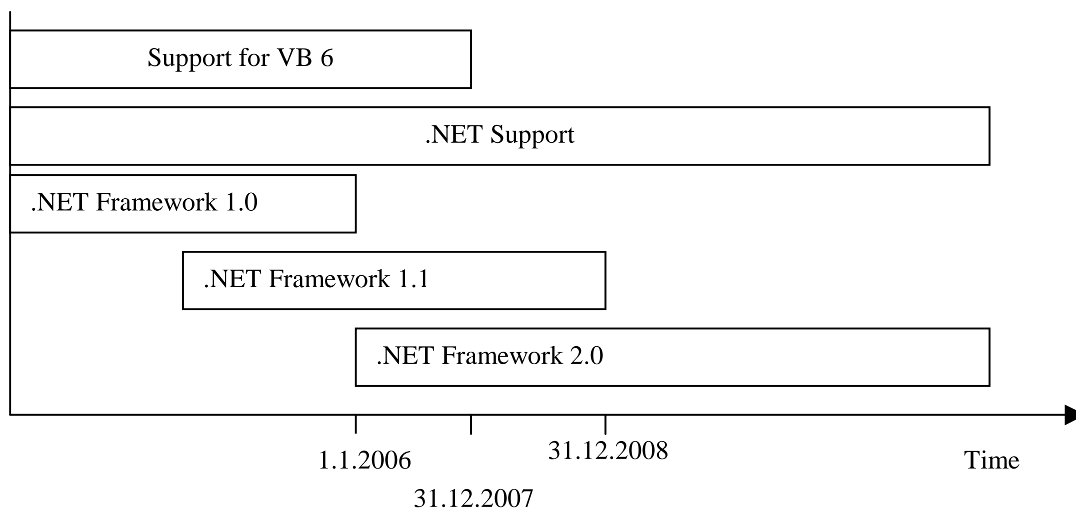
- Operating systems for the Server and Client Platform
- Office Applications
- Major business applications like ERP Systems
- Databases
- Development tools

The roadmap should span a timeframe in the future. Depending on the technology it should state a concrete decision for a technology in the foreseeable timeframe. Beyond this timeframe, when no clear decisions are possible, the roadmap should be either left open or state viable alternatives. When needed organize the roadmap in hierarchical way so that detail decisions are subordinated to top level decisions.

Who is responsible for the roadmap? Technical knowledge of products is necessary to estimate

the impact of a new technology, knowledge of the current landscape of systems is required as well as knowledge of upcoming business requirements. Therefore a broad knowledge of the business combined with in-depth knowledge of upcoming technologies is necessary. This knowledge is seldom concentrated in single persons and yields for a team to define the roadmap. The effectiveness of a roadmap can only be assured if enough management support is available to enforce adherence to the roadmap. So the IT policy unit is a good choice for the roadmap of an IT-Department.

Example:
The following image shows a possible Roadmap for the use of the .NET framework. In this case maximally two versions of the framework are supported. There is no statement on the successor of the .NET Framework 1.1 yet, because no decision has been made which version of the framework to use afterwards.



Resulting Context
With a TECHNOLOGY ROADMAP in place decisions on the supported technologies and applications based on them can be drawn on a documented base covering a defined period. Also the end of lifetime for technologies is visible so that an appropriate reaction is possible and the migration or redevelopment of applications can be initiated early enough.

Known Uses
Within Munich Re a TECHNOLOGY ROADMAP is used for strategic products.

1:1 REPLACEMENT

Limit requirements for a new system by just replacing the functionality of an old system in contrast to implementing new features.

Context:

A software system shall be replaced by a new system and you want to elicit the requirements for new system. New business requirements or technical limitations of the old system drive the development of a new system. Additionally management, users and developers of the old system know areas of improvement like features and functions that 'should be done by far better than in the old system', interfaces 'that should expose the whole functionality of the system through an API', an ‘unlimited undo/redo’ or similar.

Problem:

You have problems to limit the requirements that should be addressed by the new system replacing a predecessor system. You do not know which improvements over the old system seem vital for the new system. Sometimes you want to overcome all the deficiencies of the old system - which can end up in a system that is even messier than its predecessor, that what is called the ‘second system effect’ [Brooks].

Solution:

In the new system replace the functionality of the old system one to one. The key idea is that the development of the new system is driven by replacing the features of the old system rather than implementing new features.

Forces:

1:1 REPLACEMENT implies that you do not implement new requirements at a first glance. Though often only new requirements justify the development of a new system. From a business perspective the benefit of replacing an existing system with a new system that does exactly the same seems little. Such development effort will only be founded if other reasons. Technical limitations of the old system, safeguarding knowledge from developers who will retire or a change of platform may justify such an approach. These scenarios are suited best to apply the 1:1 REPLACEMENT pattern. Still the pattern can be applied in the development of a new system with altered requirements.

In an agile development environment change requests are welcome and are likely to dilute a strict replacement approach. In such a development environment the features of the old system can be replaced in the first set of iterations and new requirements can be addressed in subsequent iterations. Favoring the replacement of the old system towards a new usable system over implementing new features should drive the requirement prioritization and development process when applying the pattern. What about architecture and design when using this approach? As Fowler pointed out agile approaches to design do not have to deal with all design and architectural questions in the beginning [Fowler]. Designing and refactoring can accompany the whole development process.

In larger systems it can become a tedious duty to elicit the features that are provided by a legacy system. In some cases algorithms and usage scenarios will be difficult to discover, in other cases solutions may have been chosen which can be implemented in a totally different way. The predecessor system may use solutions that do not seem appropriate for a new system, it has errors or poorly implemented features. Tree-like controls in user interfaces for example are relatively new for navigation and data exploration, i.e. they are in place since one or two decades. When tree-controls offer a better alternative to work with application data they should be used for that. The

idea of the 1:1 REPLACEMENT pattern is to offer the same set of customer available functionality but with the means of a new technology. Of course, when it turns out that features of the system are never used, these need not to be implemented.
One risk of the 1:1 REPLACEMENT pattern is to fail with the implementation of new additional features. This risk is real as it is real for a system that tries to implement a greater feature set with the same technology. When you plan to change the development paradigm, e.g. using model driven development for a known technical domain in favor of just writing C-code new possibilities will arise that could never be achieved with the predecessor. This shows an other risk of the pattern, namely that thinking in features of the old system can make you blind for other solutions.

Resulting context

The newly implemented system offers the same feature set as its predecessor. When well designed the new application is better maintainable than the later. It allows deployment of the application at an earlier stage than an application with richer features.

Known uses

The 1:1 REPLACEMENT pattern leads the implementation of a system within Munich Re, where an Visual Basic 6™ - Application is redeveloped with .NET. The existing system and its developer serve as source for requirements that are implemented with a new technology by a team. The requirements for the new system are described in detail in use cases that serve as base for the implementation. As a side effect this effort facilitated finding bugs in the old system.

References:

[Brooks] Brooks, Frederic P., *The mythical man month*, Addison-Wesely 1995

[Chapin] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, Wui-Gee Tan, *Types of software evolution and software maintenance*, Journal of Software Maintenance and Evolution, Wiley, 2001, Vol. 13, 3-30

[Fowler] Is design dead ?, http://www.martinfowler.com/articles/designDead.html

[Nikora] Allen P. Nikora and John C. Munson, An approach to the measurement of software evolution, Journal of Software Maintenance and Evolution, Wiley, 2005, Vol. 17, 65-91

[Rhagozar] Masued Rahgozar, Farad Oroumchian, An effective strategy for legacy systems evolution, Journal of Software Maintenance and Evolution, Wiley, 2003, Vol. 15, 325-344

[Sahin] Izzet Sahin and Fatemeh 'Mariam' Zahedi, *Policy analysis for warrenty, maintenance, and upgrade of softare systems*, Journal of Software Maintenance and Evolution, Wiley, 2001, Vol. 13, 469-493

[Swanson] Swanson EB, *The dimensions of maintenance*, Proceedings 2nd International Conference on Software Engineering. IEEE Computer Society: Longbeach CA, 1976; 492-497

[Visaggio] Giuseppe Visaggio, *Ageing of a data-intensive legacy system: symptoms and remedies*, Journal of Software Maintenance and Evolution, Wiley, 2001, Vol. 13, 281-308