A Few Patterns for Managing Large Application Portfolios

Wolfgang Keller
c/o Generali Office Service & Consulting AG, Kratochwjlestr. 4; A 1220 Wien, Austria
Email: wk@objectarchitects.de
http://www.objectarchitects.de/ObjectArchitects/

Abstract

Merger mania hits the corporate world as a side effect of globalization. Mergers of their information systems follow most mergers of companies in the same industries. This paper presents some frequent problems and frequently used solutions (patterns) to help you manage such information system mergers. The examples are mostly taken from „elephant mergers" of corporations with application portfolios of well over 100,000 function points for the core systems alone. Such marriages typically appear in the financial industry. The patterns are also useful for companies, which have more than one information system for the same business function and want to consolidate their systems.

Introduction

Mergers and acquisitions are one of the predominant phenomena in our new globalized economy. These trends plus the urge to maximize shareholder value have created a new sport for IS professionals: Merging information systems and managing large application portfolios. This sport appears in various industries. This paper will use the insurance and banking industries as running examples, as the author knows these best. Therefore the life examples and known uses are taken from these industries. Nevertheless, the ways and methods how to manage the mergers of IS landscapes in other industries might be pretty similar to those presented here.

To give you a feeling, whether you should continue reading, imagine you are in the following situation. You are in charge of managing all the applications of a multi-national insurance group. You have an eye on and decision power for the new information systems and you also supervise the maintenance of the existing operations. Hundreds of programmers work to maintain that portfolio and the portfolio has a size of more than100.000 function points. Your shareholders and your bosses have recently got notice that you maintain as much as 13 different systems for life insurance alone in the group (which was a result of mergers and also a result of time to market decisions in the past). Now they want to buy yet another competitor

about the current size of your company and will ask you to reduce IS costs dramatically at the same time. Tough job. What usually follows is a year long clean up process and you can be sure that the next merger will appear before you have cleaned up the consequences of the last one.

General Forces

You will find that there are a lot of quite conflicting forces and interests at work here. They range from technical to domain specific to socio-cultural forces. If you manage a large application portfolio you will deal with all of them.

Cost: is about the single most important force (you might think) in the application portfolio management sport. It seems obvious that it is cheaper to have one life-insurance system in your company than let's say 13. However, this is not always true and cost is by far not the only factor that needs to be taken into account when making decisions about the future of your system landscape. Costs come in such facets as cost to create the system, maintenance costs, total cost of ownership and many more. Costs are a counterforce to Function and nonfunctional requirements such as performance or maintainability.

Function: Your systems should provide business functionality. The requirements gurus will tell you that your systems should do, what the users want them to do. But there are more stakeholders here. Maybe a board member has some ideas of what a system should do in a few years, maybe the public has an opinion about what a system should not do in terms of privacy – anyway: All of this will be summed up under the term *functional requirements*. Function is often a counterforce to nonfunctional requirements such as performance. If you concentrate primarily on functionality you might also loose a grip on technical quality.

***Nonfunctional Requirements*:** lead to system properties such as performance, security, maintainability, flexibility and the like. This will not be discussed in more depth, as it need not be referred to these forces in the later discussion. Most books on software architecture like [Bas+98] contain elaborate lists of such forces. The better you ate at these the more you pay.

Technical criteria: In order to make sound decisions you also need to check the technical quality of the systems you want to make plans for. Criteria include: "How complex is a system compared to the simplest system that possibly does the job", "Does the system fit the overall technical architecture strategy (if you want to build web apps it is not a good idea to buy a new fat client system or to even keep one for all eternity) and many many more.. These items are in most cases a counterforce to costs and rich functionality. The cleaner you want to become the more you have to invest in redesigns and rework.

Socio-Cultural Forces: Besides the hard technical criteria, functional criteria, and nonfunctional requirements there are a lot of soft forces. If you are dealing with reuse you will have heard of the NIH (Not Invented Here) syndrome. There is also usually pride of the developers who own and maintain a system, and there is a general attitude against change in many humans (people hate change), there are various political forces like "what will happen to

if my system is abolished and the data are migrated to other systems?" and more. You will find most of these in books on general change management. If you ignore these forces, you will typically have to fix a lot of problems with money – so this is also related to the cost force.

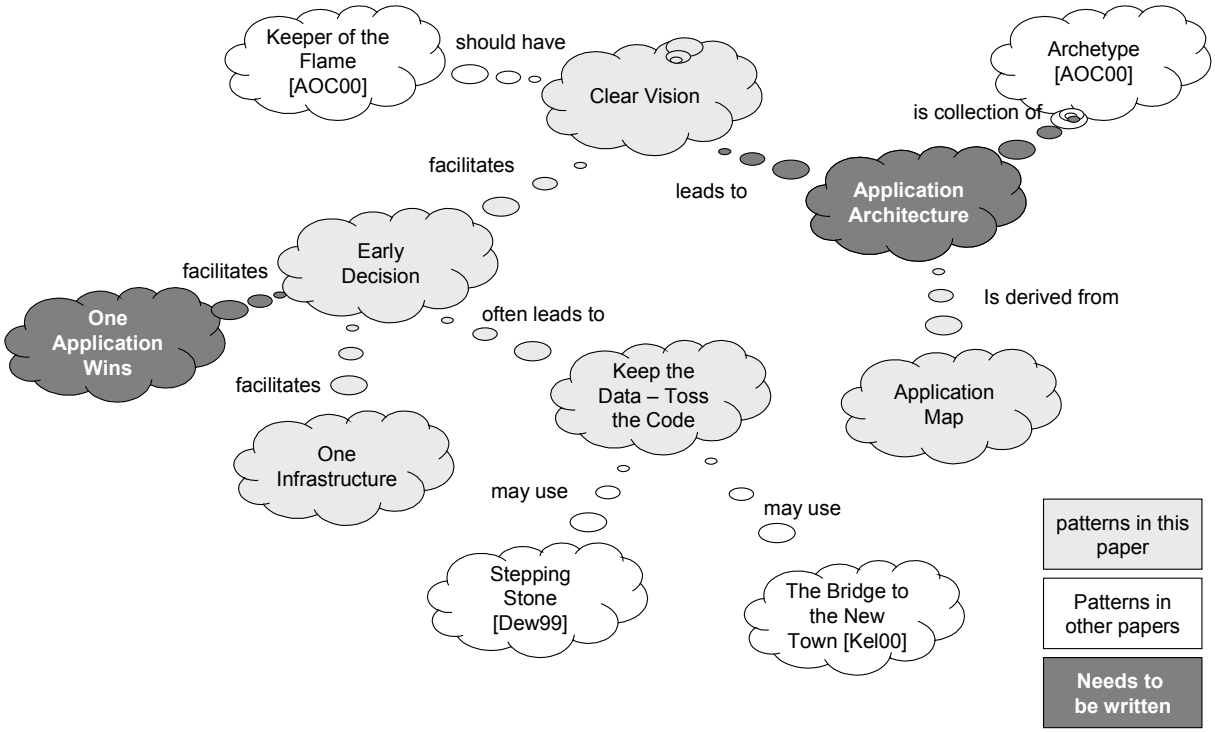
Time to Market: Besides all these internal forces we did not yet talk about the customers – the ones who bring the money and the ones we want to provide with up to date products. It is very interesting if the EDP (Electronic Data Processing) department has optimal operation costs – but if it supports products that have no positive resonance on the market, the company will have trouble paying even the lowest bills for the IT department. So if you look at it from a bird's perspective, it might often be good for the whole company to accept a few extra costs in the EDP department while selling more and having a better bottom line. Time to market is a counterforce to almost any of the above forces. Often time to market will force you to spend a lot of money on a crappy solution just in order to be present on the market before others are.

Project Risk: Another important factor is project risk. If you have an ideal solution that has an 80% risk of failure due to a lack of relevant skills or because it is simply a death march project, you might want to favor a solution, which has only 70% or 80% of the features but only a 10% risk of project failure. You might even plan parallel strategies like doing the low risk solution first and then trying the high risk one. Time to market often forces you to take big risks and if you have cost cutting expert as your boss this will also often increase risk.

As you see, optimizing an application portfolio needs to deal with a lot of conflicting forces. Every force conflicts with almost all the others. A good application portfolio manager will have to deal with many of them without loosing focus and without using them as an excuse for total chaos. Evaluating a solution here often means dealing with soft and political factors, which are very hard to quantify. Therefore you will often have solutions and plans that have similar measurable costs but you need to decide anyway and might have a few more enemies afterwards.

Roadmap

The roadmap gives you an overview of the five patterns presented in this papers and relates them to other patterns in the area.



The relations will become clearer from the related patterns sections of the patterns. Once you have read the patterns this roadmap will help you even better to relate the patterns.

The Patterns

Pattern: Keep the Data – Toss the Code

Also Known As

Another term for the same thing is “Data Migration”. “Keep the Data – Toss the Code” is a motto from a book by Michael Brodie and Michael Stonebreaker [Bro+95].

Example

Imagine you have two (or more) information systems with very similar functionality. They are called A and B. For cost reasons you want to replace them with only one system. Each of the systems covers about 70% of your desired target system’s functionality – but the 70% are different for each system.

Problem

What do you do, if you have two or more systems with almost identical functionality and want only one system?

Forces

Cost: Why not keep the two systems? That depends on your cost situation but in general maintaining two systems on the long run is more expensive than migrating to one new system.

Technical quality of the systems involved: For your decision what to do, you will evaluate the technical quality of both systems. You will ask questions such as: "How old are the systems", "how many function points do they comprise?", "how complex are the systems that support the same functionality?" and more. Other criteria include performance, or the technical environment used.

Functionality: As mentioned above both systems, A and B do not have the desired functionality. So why not use the best features of the two of them and merging the two? Or you might also pursue the idea to write on new system instead.

Politics: Often the choice of a new system will be influenced by who is more powerful: The owners of system A or the owners of system B. Be aware that any decision that will be made is not free of politics. Identify the stakeholders in your scenario.

Time to Market: Any migration puts your EDP department's capability to support new products to a rest for some time. So if you don't want to loose market share, you might have an urge to do a fast job here.

Solution

Keep the Data – Toss the Code.

migrate the data from all the systems you have into one common system.

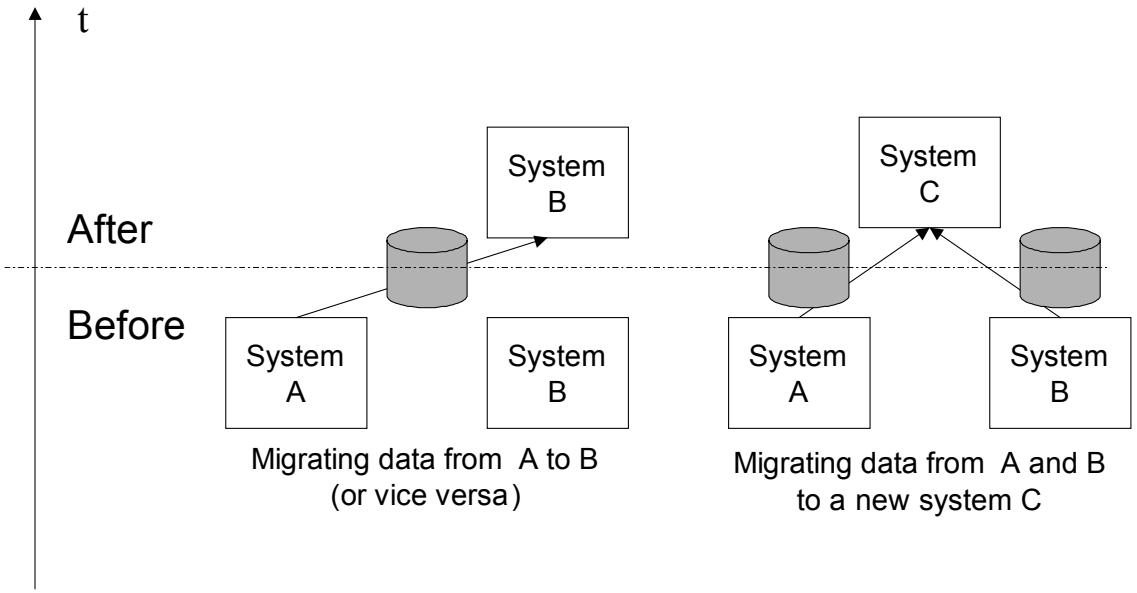


Figure 1: Data migration scenari

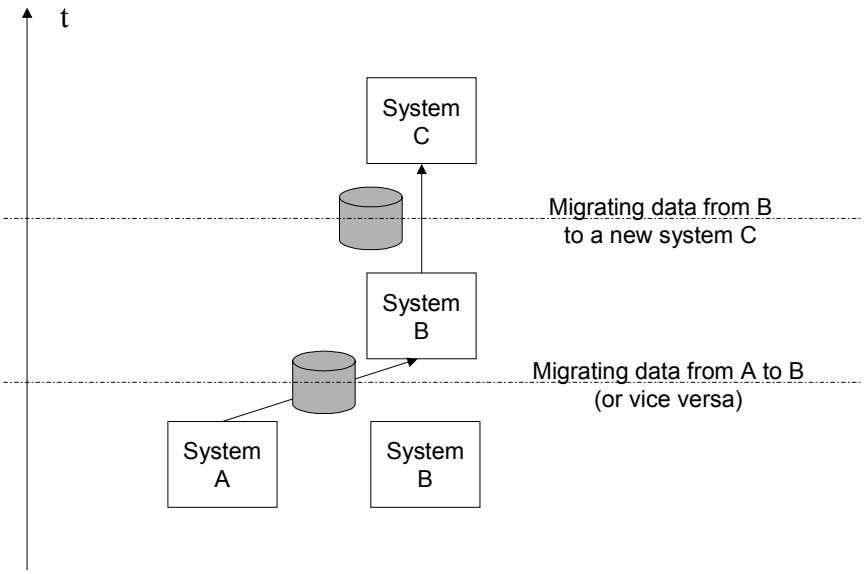
You can migrate the data from system A to system B or vice versa or you can write a new system C and migrate all the data to system C, but you should not try to use a reengineered code mix of system A and system B as your target scenario.

Variants

As we pointed out in Figure 1 there are two major variants here:

- Migrating the data to one of the existing systems,
- or migrating them to a new system, which has to be created in addition to the two existing systems.

In practice you will seldom find one of these solution types in pure form. You will instead combine the first type with some functional enhancements of the target system or you could even use one of the old systems as an intermediate system while you build a better system C as illustrated by the below figure.



Consequences

Technical Forces: Why is reengineering and combining "components" from two systems a bad idea?. Well – if you have perfect systems and if system A and system B are based on similar or the same component technology reengineering might be a viable way. This kind of reuse is the goal of component based systems. However, in today's world, mergers seldom have to merge software based on such component technologies, certainly not on the same. It is more likely that you will find one COBOL system let's say on CICS as system A and one PL/I system on IMS as system B. Combining the two is possible – in theory – but too expensive in practice.

Cost: Your new system portfolio (let's say B only) should be cheaper to maintain than A and B together (see Example Resolved).

Functionality: The new system will still not have 100% functionality, but let's say still the 70% ob B before the data migration. If you want more functionality, you need to start a new project to enhance B or even migrate it to a new system C.

Politics: Whatever the solution is – it will be one that is accepted by those who had the power to push the new solution. There will be some "losers": The users of the abandoned system and its developers. These two groups need to learn new stuff. One group will also be offended, as their system is dumped. Hopefully there's a change management process that makes as many people as possible winners. The point here is that you make only about 50% or less of the people unhappy. If you try to find a political compromise that makes close to 100% of the people happy, you shareholder might be very unhappy, as such political compromises are often very expensive on the long run.

Time to Market: Any migration will need time and will consume resources. People cannot implement new products while they migrate from old system A to old system B. Therefore don't expect the business departments to support you luckily as they have no additional

functionality, no added value for the customer but immense costs for explaining the old stuff to developers, and for testing. All this does not improve their business situation as no new functionality is being is being implemented.

Risk: Compared to any form of code reengineering data migration is a low risk strategy. Compared to building new systems they are also low risk strategies as there are often people who know the target system and know how to handle it. But remember: No risk, no competitive advantage, and no profit. If you keep migrating your old stuff to death a new competitor with new systems might give you are very tough time on the market.

Related Patterns

In order to evaluate the systems that you want to consolidate you need a Clear Vision. This vision will usually be expressed in an Application Map, which is your overall building plan at a city level, if you consider single systems the "houses". If you have to apply the pattern in a merger situation of two companies, you should not try to make hyper-accurate decisions. Try to make Early Decisions instead. Using the pattern will lead you towards One (technical) Infrastructure.

Often people use The Bridge to The New Town [Kel00] (a.k.a. Live & Let Die [Dew99]) to thin out a legacy system (in the above case system A). The variant above has large structural similarity to the so-called Stepping Stone pattern [Dew99]. System B is only an interim package that shall finally be replaced by system C.

Known Uses

The known uses for this pattern are TNTC[1]. Generali Vienna Group uses the pattern in the so called EAS project. This project merges the two legacy application landscapes of Interunfall and Generali Austria. AMB Group Germany uses the pattern for system mergers between it's group members, AXA insurance group use the pattern in order to flatten bought companies, most banks use it, e.g. in the Hypo Bank, Bayerische Vereinsbank merger.

Pattern: Early Decision

Also Known As

One Application Wins sooner or later. You will see why sooner is better than later ☺

Example

Imagine you have just heard the announcement of the merger of two large financial institutions – let's say banks. You have been appointed the new CIO of the joint AB-bank and you need to come up with a plan, how to join the two systems. As you are from one of the two banks, let's say the A-bank, you have a certain hate-bonus from the B-bank's former employees who want to see their system win. As you do not want to loose the B-bank

[1] too numerous to count

employees you think it is a good idea to start a system selection process with objective and measurable criteria. For each of the subsystems you start workshops and compare and evaluate the subsystems. The process takes one year and after that process you migrate the core banking app to the A-bank's app, the security trading app to the B-bank's solution and so on and so on. Finally you have spent a year in workshops and have the entire project before you.

Problem

How do you avoid a long and expensive evaluation process when you want a decision for one out of several very similar software systems?

Forces

The forces that apply here are the same as the general forces above, plus the forces from Keep the Data – Toss the Code. Except that there is often an extreme emphasis on politics involved here. All the people in the process want their system to win and will find whatever "objective" reasons needed to support her opinion.

Solution

Make an Early Decision. Decide fast and hard …

and avoid too many workshops as these are seldom really "objective" but are more often used to express the political will of the participants and to pull marketing stunts.

Example Resolved

Depending on your overall situation there are many good solutions – we will just give a few examples here:

- In one merger of equals of two big German banks the decision was made in favor of the system of the bank that was "the more equal one" in the merger of equals – despite the fact that the system was somewhat technically inferior to the other bank's system.

- A large European insurance group specializes in taking over smaller insurance groups. They have made a professional exercise out of this and have reduced the time needed for a merger of all EDP systems from about three years to about 9 months.

- Another German insurance group regrets that they did not act quickly. They had a merger of more or less equals at least of companies of similar size. They had the array of workshops and a system landscape that is a mixture of many worlds and produce the appropriate amount of trouble.

- In a merger of two Austrian banks the larger bank adopted the smaller bank's system because it was simply better from a technical and functional perspective. But they knew what they wanted to do before they bought the smaller bank. This did not prevent some of the owners of the small bank's system from being pushed out of management functions once they had done the migration job. Bad luck.

Consequences

Cost and Function: Acting quickly may come cheaper than expensive and long-running pre-studies. The situation that each system covers about 70% percent of the desired functionality, as described in the example for Keep the Data – Toss the Code is rather typical.

Nonfunctional Requirements and Technical Criteria: Can become a killer, if one of the companies that merge has a significantly better system than the other one. In this case there might even be a quick decision for the smaller partner's system that is otherwise "less equal" but simply has the better system.

Politics: If you decide quickly following the powers you have the least political trouble. You might loose a few people, but you will win a lot of time and save a lot of money. A quick decision implies the danger of choosing the wrong application from a technical point of view. But in most cases when you have to make such a decision, the solution is somewhat clear. In most cases you will find two systems that have roughly the same degree of maintenance and functional problems it does not really matter which on you choose. So you make a fast decision with regard to politics. If one system is clearly superior, you can also afford a fast decision against a tougher political force because you have clear fact on your side.

Time to Market: As you avoid the paralysis of a long shoot-out and comparison process you have a much better time to market and you can react to the market much faster than with a long slow evaluation process.

Project Risks: As you have experts for all systems you want to merge and as you will likely keep the experts for the winning system, a quick decision should be neutral in terms of project risk.

Related Patterns

Once you have an Early Decision the next stop will very likely be a Data Migration (Keep the Data – Toss the Code). If you avoid the cascade of workshops and make a clear cut, this will most likely go together with One Application Wins and will result in One Infrastructure.

Known Uses

See the Example Resolved section above.

Pattern: Clear Vision

Example

Imagine you want to organize the application portfolio of an insurance company. It is of some importance that everybody uses at least similar construction principles and ideas over the whole application landscape in order to avoid double implementations of functionality and in order to make the weakest link of the chain (the whole application portfolio) strong enough for your customer's goals.

Now imagine you have to communicate a 127 slides detailed building plan in order to communicate your idea of what the insurance system landscape should look like.

Problem

How do you communicate architecture for a whole enterprise application portfolio?

Forces

Simplicity versus accuracy: It is a good idea to have a simple vision but it may be very tough to implement it – a simple vision alone does not tell you how to implement it. If you look e.g. at the NASA's former vision "put a man on the moon" this will help you to judge whether some action puts this goal forward but it will not yet give you a clear blueprint.

Solution

Formulate a Clear Vision …

that allows you to put the core construction principles of the whole portfolio on a single slide or two at most. Well you might say "Smart A.." but where does such a vision come from – it often evolves in a domain and you only need to pick it up. The vision should be driven by business needs.

Example Resolved

For example in an insurance the vision might be, that you want a system landscape that

- Allows flexible product definition (short time to market),
- allows you to regroup process steps to ever new workflows (business process orientation),
- allows you to use the system for more than one subsidiary,
- is object-oriented for better maintenance and therefore lower maintenance costs.

No matter whether you think that this is the correct vision for insurance systems, you can look at a system and tell whether it is conformant to it or not or in a way conformant. What you need in addition to that is a building plan that follows the "Mile Wide – Inch Deep" [AOC00] pattern and tells your coworkers something more about how to implement the vision. But still "Smart A…": Where do you pick this up. We can only give the answer for the insurance industry. The above construction principles have evolved over 15+ years as folklore in the industry – if you look at other industries they also have their Domain Architectures and high-level domain principles. So the solution could also be … pick up the industries vision and formulate it in a clear way.

Consequences

The clear vision is clear but it will often not tell you very much about how to implement a system that makes your vision real. Hence you need some complementary material in order to really communicate your enterprise architecture. Such material could be a top-level architectural plan plus a Domain Architecture that comprises all the subsystems. But no matter how deep you are involved in the design of a single subsystem. Often you can still ask – is what I do here good to achieve the vision?

Related Patterns

The pattern is closely related to the Keeper of the Flame [AOC00]. The clear vision is necessary – but a Keeper of the Flame is also needed to enforce the integrity of architecture. In a lot of decision situations you need a clear vision to be able to judge the quality of the applications in your Application Map. The Domain Architecture will typically express your Clear Vision and detail it at a lower level.

Known Uses

Generali's Phoenix program has a clear vision, which has been valid for over ten years now and still gives a stable guideline on how to build applications. Other insurance architectures such as VAA (Versicherungs-Anwendungs-Architektur see http://www.gdv-online.de/vaa/) or IAA (Insurance Application Architecture by IBM) have very similar visions.

Pattern: Application Map

Example

Imagine you are the CIO of an insurance company that has grown historically across half Europe. Sometimes new systems come into the portfolio of your company when your company buys a new subsidiary. Sometimes you get a new app when there are market needs for new products. You have quite a few apps for the same functionality in your company; let's say 10+ life insurance systems. Everything has grown historically – but you need to clean up.

Problem

What is your basis for planning what software is installed where and what software will be installed where?

Forces

Simplicity, cost and understandability versus accuracy: A planning methodology needs to be usable in a sense that you need to come to a result in a finite amount of time. If you look at the size of portfolios we have to deal with (100,000+ Function Points), this is not trivial. So you have to find a simple instrument that is good enough, cheap and easy to understand by all stakeholders

Solution

Use a planning instrument called application map.

You will analyze the application portfolios for all your subsidiaries. This will result in a series of application maps. You will then think of how to reduce the number of apps, taking into account the general forces.

Structure

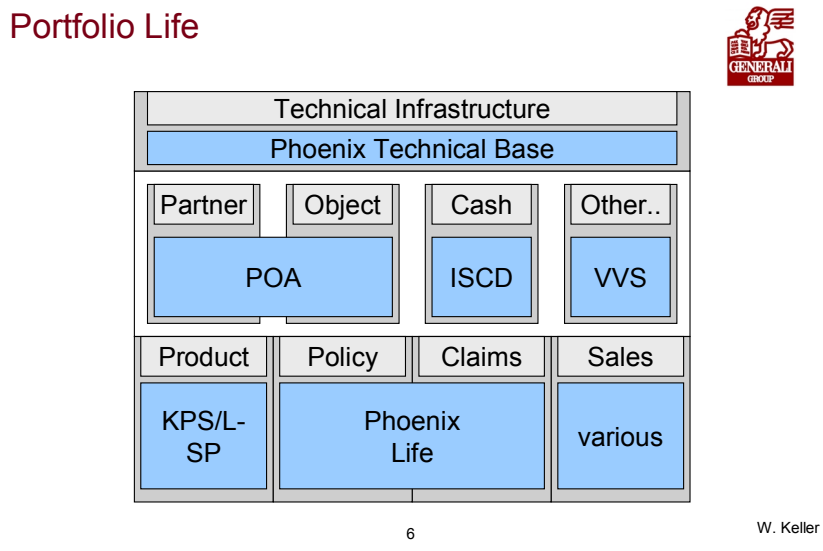


Figure 2: Sample application map – real life example

The structure of an application map is typically derived from Domain Architecture. So for each major app (or call it package in the domain architecture) you say what covers this part of the domain architecture in the application portfolio of a single company. E.g. in figure 2 the system POA covers the Partner and (insured) Object parts of your domain architecture. So what you do for each application is you check which part of the domain architecture is covered by its functionality. Often this will not result in such nice pictures as figure 2 above. For example an app might have functionality that is cluttered across or parts of the map might redundantly be covered by two or even more apps.

Example Resolved

For the job of application portfolio management you have a few instruments that help you:

- The above application map,
- Your Clear Vision helps you detect what is "good" and what is "bad" in a sense of what helps you to come closer to your overall vision.
- And your Domain Architecture provides you with the domain knowledge necessary to judge the applications.

Consequences

Simplicity versus accuracy: Experience shows that the mechanism is usable and yet precise enough for real world planning problems. It is also simple enough to allow non-techies to understand it.

Related Patterns

Application Maps help you find proper Early Decisions. They are usually derived from a Clear Vision via the Domain Architecture.

Known Uses

Application maps are widely used. As you can see, we use it ourselves. This is no strong evidence. Aachen Münchener Group uses very similar maps. SAP is using so called industry solution maps that tell you which SAP solutions to use for which industry. This is indeed a stronger known use as SAP uses those extensively.

Pattern: One Infrastructure

Example

You are again the CIO of the newly merged AB-bank. You have ignored the Early Decision pattern and have decided for a mixed application portfolio that contains the best from A-bank and B-bank. Nevertheless you Kept the Data and Tossed the Code in order to not add additional complexity. After a one-year shoot out process and two years project run time you have a single application landscape. But you hear more and more complaints about people who have to be familiar with two print systems, two archive systems, two authorization systems and so on. You find that you have two suites of technical infrastructures[2] in your portfolio – the one from A-bank plus the one from B-bank,

---

2 the term „technical infrastructure" denotes all the technical helper applications like a print system, a workflow system and the like.

Problem

How do you choose the appropriate technical base infrastructure if you have an application portfolio from several sources that come with more than one infrastructure?

Forces

The forces here are the same as for Early Decision

Solution

Choose One Infrastructure and not a mixture of components from two infrastructures

Make an Early Decision about this one and ignore negative comments from one of the two sides, because if you do not decide for one, you will hear the complaints three years later, will have the cost and will have a much harder time improving the situation right away from the start.

Example Resolved

We can revisit the examples from Early Decision.

- In one merger of equals of two big German banks the decision was made in favor of the system of the bank that was "the more equal one" in the merger of equals – despite the fact that the system was somewhat technically inferior to the other bank's system. As a side effect these two banks choose **One Infrastructure**.

- A large European insurance group specializes in taking over smaller insurance groups. They have made a professional exercise out of this and have reduced the time needed for a merger of all EDP systems from about three years to about 9 months. They do not discuss – they have **One Infrastructure**.

- Another German insurance group regrets that they did not act quickly. They had a merger of more or less equals at least of companies of similar size. They had the array of workshops and a system landscape that is a mixture of many worlds and produce the appropriate amount of trouble. As a result **they have Two Infrastructures today**, leading to the above complaints.

- In a merger of two Austrian banks the larger bank adopted the smaller bank's system because it was simply better from a technical and functional perspective. But they knew what they wanted to do before they bought the smaller bank. As a result they also have **One Infrastructure.**

Consequences

The consequences are similar to the consequences for an Early Decision. Please note that One Infrastructure might be somewhat expensive if you have to use one or more applications that are based on just the other technical infrastructure for domain reasons. In this case the decision is driven by the costs for not having the better application for your business versus

the costs of having to maintain two technical infrastructures versus the cost of reengineering the application.

Cost and Function: In many cases both alternative technical infrastructures are almost identical from a functional standpoint – but very different from a technical standpoint making interoperability an expensive hobby. Therefore a decision comes cheaper and will seldom do any damage expressed in reduction of available functionality.

Nonfunctional Requirements and Technical Criteria: Can become a killer, if one of the companies that merge has a significantly better system than the other one. In this case there might even be a quick decision for the smaller partner's system who is otherwise "less equal" but simply has the better system. Same as in Early Decision.

Politics: Even techies are full of politics. So expect the most curious arguments for either of the technical infrastructures to choose from.

Maintenance Costs: as you have to maintain two infrastructures, have to buy two sets of licenses, have to install two upgrades instead of one, the costs are in most cases much higher if you do not use the pattern.

Usability: Your users will force you sooner or later to move towards one infrastructure. So why not do it right from the beginning. Users don't want to understand why they have to use two desktop calendars instead of one, two or more different workflow systems and the like. So there will be pressure to install one infrastructure later anyway.

Project Risks: As you have experts for all systems you want to merge and as you will likely keep the experts for the winning system. Any decision does.

Related Patterns

One Infrastructure is usually the result of some form of an Early Decision; leading to the fact that One Application Wins as a complete application resulting in the least trouble. One Application will Win sooner or later. The earlier your decision the better for your cost structure.

Known Uses

See the Examples Resolved section above.

Credits

Thanks are due to various persons. Harry Fräser has developed the Generali application portfolio management method. Thanks are also due to Rick Dewar and Alan O'Callaghan for their related patterns. Special thanks go also to my EuroPLoP shepherd Markus Völter who has invested large amounts of time and diligence in improving this paper.

References

[AOC00] Alan O'Callaghan: Patterns for Architectural Practice, Proceedings EuroPLoP 2000, to appear, preliminary version can be found at http://www.coldewey.com/europlop2000/papers.html

[Bas+98] Len Bass, Paul Clements, Rick Kazman, and Ken Bass: Software Architecture in Practice (SEI Series in Software Engineering), Addison-Wesley 1998.

[Bro+95] Michael M. Brodie, Michael Stonebreaker: Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach; Morgan Kaufmann 1995.

[Dew99] Rick Dewar: Characteristics of Legacy System Reengineering, pattern Writing Workshop, EuroPLoP 1999, Bad Irrsee, Germany. See http://reengineering.ed.ac.uk/publications.html for an online version.

[Kel00] Wolfgang Keller, The Bridge to the New Town: A Legacy System Migration Pattern, Proceedings EuroPLoP 2000, to appear, preliminary version can be found at http://www.coldewey.com/europlop2000/papers.html