The Bridge to the New Town

A Legacy System Migration Pattern

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

Pattern: The Bridge to the New Town

Also Known As

Live and Let Die [Dew99]. One out of many variants of "Keep the data – Toss the code" [Bro+95]

Example

Imagine you are the person responsible for the application portfolio of a large insurance company. For their business the company runs one monolithic system for most of its product divisions[1]. This system is 25 years old, written in Assembler and sits on an IMS/DB database. *This is the old town. It's at the end of it's potential growth and people want to move out of it to the other side of the river*. Let's call this old system (*or town*) CLS for complete legacy system. You want to replace this system with newer systems, *a new town*, in a step by step fashion, *as you cannot afford to build the new town in a single effort. So you will not move into the new town in a single move but step by step as new buildings get ready.* The first new

[1] Please note: This is NOT the current situation of my company, but an older example (4 years old) of one of the companies that were merged to the current Generali Group in Austria.

system to get ready in a new modern client server architecture (*first new building complex*) is a business partner system – lets call it POA (for partner object address). This new system is a 2-tier client/server system using DB2. In the old system CLS there's a part that can be somehow located that deals with entities (persons and their addresses) that are very similar to the ones covered by the POA replacement system.

Problem

How can you replace a part of an existing monolithic system with a new component or a complete new application portfolio.

Forces

***Speed of Migration versus Risk, Size and Complexity of the Legacy*:** It could be one idea to migrate not only a part of the existing system but the whole system in a single step. This would be called a Big Bang Migration strategy. Big bang strategies have the potential of being faster and cheaper than any strategy that works in a stepwise manner. Big bangs tend to be very risky – if you fail, you fail completely. You have the legacy as your parachute but it the legacy cannot survive a certain critical date like e.g. Y2K or the Euro introduction you will have a 100% solution or a 100% failure and you know after a let's say two years project at the very moment you try to switch on the new system. If the legacy (and also the new system) is rather big and complex, big bangs are even more risky. For the same reasons people like incremental development better than the waterfall model nowadays. If you work in an incremental fashion, you get feedback earlier. Therefore it is a better idea to proceed in little steps, even if the way takes a little bit longer than an optimum big bang. Because the risk of a failure is too high. On the other hand, if you have a very small legacy that can be maintained by a team of only a few people, a big bang might be the cheaper solution and safe enough. Size of the legacy is really an important factor. *You will not renovate a very large and complex building in a week or even in a few years, but this is a permanent activity. On the other hand you are able to repaint a family home in 2 days or you are able to completely replace it with a prebuilt home in a few weeks.*

Layered Legacy versus Spaghetti Check In/Check Out Persistence: Your migration job will be much easier, if your legacy has some structure like a layered persistence architecture (data are clearly separated from the code) or a clear modular structure. Your migration job will be tougher and your risk for a stepwise migration will be higher if you have a big ball of spaghetti or mud, where it is tough to find the code that deals with data or where you check in a whole object net at a time and then check it out upon leaving your program. Stepwise migration schemes have proven very effective in large, layered business systems. They are not common in let's say CAD applications that have check in / check out persistence and a tightly coupled and very woven object net.

Risk due to Constant Change: The world and therefore change in business does not stop for two years while you want to migrate or replace a system. While you make your plans and also while you perform the actual migration you will get a continuous flow of new requirements from the business people who pay your salary by dealing with the customers in a dynamic market using you system. This makes all migration more risky and also costly as changes often have to be built in two systems –the old and the new one.

***Risk versus Operation and Maintenance Costs*:** Big bangs have the advantage that you will have only one system at a time. If you use another strategy which replaces one system with another in an incremental fashion you might have the two systems (the old and the new one) running in parallel for a certain amount of time. This might mean: Double disk space, double processing costs, effort for the bridge and more.

Overall Performance and Resource Consumption versus Risk: As mentioned above, you will have only one system up and running in a Big Bang Migration, while you might have a new and the old system running in a stepwise scenario. This will have negative implications on system performance (you need two database writes instead of one) and resource consumption (you need twice the disk space, more processing power). If you say – no problem, my desktop processor is idle most the time anyway, the same is seldom true for host systems. In a host system, processing power does not come cheap and is mostly used 80% or more. If you add another lets say 40% you need to expand your machine for the period of the migration.

More Forces: Besides that, all the other standard forces, a.k.a. nonfunctional requirements in software architecture apply, like reliability, security, testability and many more. See e.g. Len Bass et al.'s excellent book [Bas+98] on software architecture for a list of such forces and a detailed explanation.

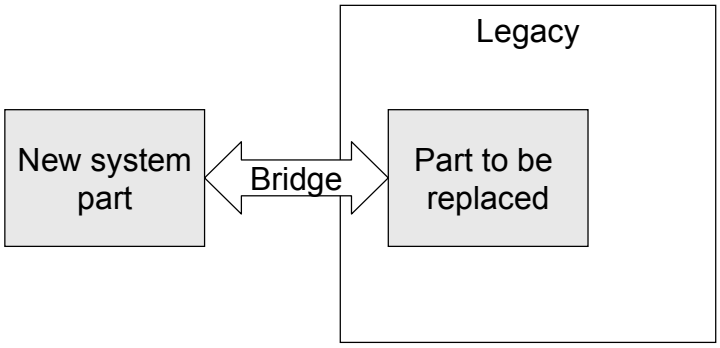
Solution

Install the new system part in parallel to the old system and connect the two using a Bridge. Cut off any write access to the old system's equivalent part (the one to be replaced) and redirect it to the new system.

Now The Bridge to the New Town connects one new building complex with the old town. – if you repeat this scheme for more than one complex of the old town, you will replace more and more of the old town by new buildings. The bridge will need to be able to handle more and more traffic. Once everybody lives in the new town, you no longer need the bridge and the old town will fade away.

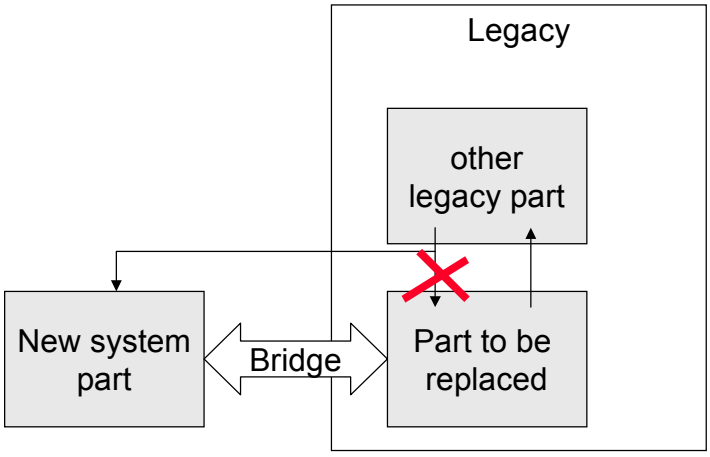
Structure

To give you a better understanding of the solution see the following sketch.



As mentioned in the solution you have the *legacy*, which contains a *part to be replaced*. In order to replace this part with a new system, you install the *new system part* and connect it to the *legacy*'s *part to be replaced* using a *bridge*.

That way the other parts of the *legacy* are undisturbed as they can continue reading data from the part to be replaced.



The above figure shows that you have to watch especially inbound data traffic into the *part to be replaced*. You need to redirect all of this traffic to the *new system part*. Either via the *bridge* or directly online. On the other hand, any *other legacy part* can work on in an undisturbed fashion as the read access to data entered in the *new system part* need not be redirected to the *new system part* but the code may still access the *part to be replaced*.

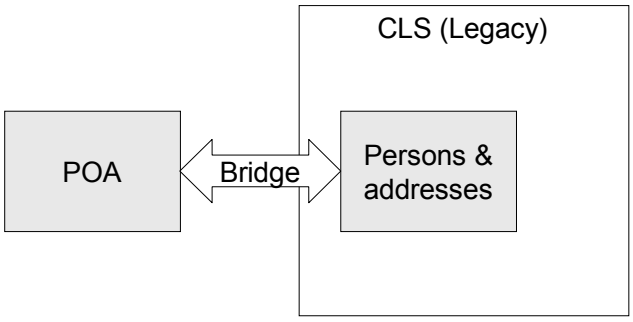
Planning Aspects

You might now ask. "How do I pick the *part to be replaced*?": Often you want to install a new system like a business partner system (in other organizations e.g. a customer database). In this case, you have to find all the pieces of the old system that perform partner functionality. If this functionality is neatly concentrated, your job will be an easier one. If the functionality you need to replace is very cluttered, you will have a harder time.

Another frequently asked question is: Why would I only want to bridge the write access and not also the read access? In most cases you use the pattern in order to replace the whole legacy in more than one step. Your goal is, to have a consistent new system and to move there with a good combination of effort and risk. If you redirect read access from the old system to the new system instead of the old parts, this is usually additional effort because you have to touch far more code than for a bridge. This is why you want to leave all those reads to the part to be replaced untouched. Never touch a running system ☺.

Example Resolved

Applied to our above example, we get the following picture:



The new system part is POA. The persons & addresses part of the legacy CLS is replaced by POA. A bridge is installed between POA and the persons & addresses part.

Variants

There are plenty of variants for this pattern. The variability comes mostly from how you build the Bridge and what data you deal with first in the process of Data Migration.

Let's start with what data you migrate first. In our resolved example you could start the POA system with a migrated replicate of all of CLS's persons and addresses. But you could also start with an empty POA database, give it a test run for some time and do the full data migration later, if you want to Start Slow.

Concerning the Bridge you have the choice of an online bridge strategy, which does instant replication or a batch bridge strategy which polls both databases for data that need to be replicated. In some cases you might also be able to live with a unidirectional Bridge . In other cases you need bi-directional data replication.

Often the new system will also contain data that have no equivalent in the legacy. In this case you may tailor the bridge to filter them out or you may also build the bridge so that it does not even read them.

Consequences

***Speed of Migration versus Risk, Size and Complexity of the Legacy*:** With The Bridge to the New Town you can profit from partial results (*new buildings*) earlier in the process than with a big bang migration but the whole way might consume a little bit more effort and take a little more time than an optimal big bang, But which big bang has ever been optimal, especially which large scale big bang? And who wants to take the risk of a total project failure to gain let's say 40% of the project costs for bridges, repeat data migrations and the like.

Risk due to Constant Change: is a challenge in all software renovation projects. This will hit you in a big bang scenario and also in a Bridge to the New Town scheme. You will always have to do double work: In your old and in the new system. You cannot evade this – you can only minimize it by speeding up the project and thus minimizing the number of changes, that are dependent on the time span you cover.

***Risk versus Operation and Maintenance Costs*:** As discussed above The Bridge to the New Town is a low risk migration strategy at the cost of some redundancy (*double buildings, the old one rotting*). If you keep up an application portfolio with a redundant system part as

produced by The Bridge to the New Town for a longer time, it is clear that you will use extra system resources, that you will have higher maintenance cost in case of changes and also higher overall operation costs than with a non-redundant system.

Overall Performance and Resource Consumption versus Risk: You also pay for the risk reduction with a decrease in performance (more disk writes due to redundancy and the bridge) and also with higher resource consumption for CPU, disk space and memory usage.

Related Patterns

There is a whole set of procedures (call them patterns) often used in legacy system replacement, which are used over and over again but which are not described as patterns yet. These are in most cases related to a Bridge to the New Town strategy, as you will inevitably use one or the other if you implement The Bridge to the New Town

- In order to keep the legacy and the new parts consistent you need a Bridge (not the bridge as described in GOF, but a bridge that replicates data). *The bridge is not for free – you have to invest in bricks and mortar.*

- Often you need Normalized Interfaces in order to build the bridge. Normalized Interfaces are also useful to reuse legacies in new web applications. *Hard to find an analogy for this one ;-)*

- If you want to dry out the legacy analogon of your new system part, you need a Data Migration. This means converting the data to the new format once. *You have to call the moving truck not only once but each time you move people into a new building.*

- Often you Start Slow with the new system part with New Products First and migrate the data for old products later. *You start with totally new buildings and move the rest later.*

- You use all these patterns to avoid a risky Big Bang Migration. *By this you avoid effects like a Brasilia from the drawing board but build a living system on the other side while learning from you problems.*

Most of the above patterns are looking for somebody to write them down.

Finally the pattern is closely related to Brodie and Stonebreaker's central advice for legacy migration: "Keep the data – toss the code" [Bro+95]. This is exactly what is done here. *You move you household to a new building and abandon the old building.*

Besides that there's plenty of reengineering patterns that deal with how to improve a system by restructuring parts of it (keep and improve the code, rewrite certain sections, refactor the system). Some of this stuff, which is already close to patterns can be found in [Brö+96] and more will also come up in the other papers of EuroPLop 2000 [will be cited here].

Known Uses

At Generali Group alone the pattern is used at two different locations for three different large scale migration projects in different context, done by different people:

- The example above is a live example of the GZAB/POA migration project at EA Generali, performed in 1996 and 1997.
- The EAS (Einheitliches Anwendungssystem) project uses the pattern to install a mixed portfolio of a legacy system and newer parts (partner, claims, commission system, others) in order to have a Euro-fit system for January 1, 2002. This portfolio will be evolved further after the full Euro introduction by further eating up the rest of the legacy part.
- The PVS/PDFS migration project at Generali Netherlands aims at replacing a complete monolithic legacy system with a set of more modern corporate system components.

Besides that there are the known uses cited by Rick Dewar [Dew99] and many projects that introduce ERP systems like SAP, BaaN or others in order to partially or fully replace a legacy system.

Credits

Credits are due to Rick Dewar (http://www.dcs.ed.ac.uk/~rgd/) for his versions of Live and Let Die [Dew99] which I shepherded at EuroPloP 1999 last year. I'd like the Generali Netherlands team for their inspiration which they called Pac Man Migration. This also would have been a good name. I'd also like to thank the EAS teams, especially Bilal Musa for discussions about the POA/KDB migration problem which brought me to write this pattern in order to show not only them that they are doing something which has been done before and which is very safe and very common. And finally I'd like to thank Steve Berczuk, my EuroPLoP 2000 shepherd for helping me make many things explicit which are clear if one deals with this over and over again, but which are no way clear *if somebody comes from another town ;-).*

References

[Bas+98] Len Bass, Paul Clements, Rick Kazman, 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.

[Brö+96] Peter Brössler; Wolfgang Keller: Wege zu objektorientierten Software-Architekturen, Proceedings GI Jahrestagung, 1996, Klagenfurt, Austria.

[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.

