Modernization Podcast #1

Mark Schroeder of SoftwareModernization.com interviews Databorough's Mark Tregear.

Automating Y2k projects was the first mainstream success story for Databorough's tooling and services. 15 years of research, design, development and product evolution later Mark Tregear reveals how Legacy applications can now be automatically modernized using Databorough's most recent versions of X-Analysis. The combination of 25 years of experience and success as one of the world's leading thinkers and practitioners in this field makes for a very interesting interview, available as an MP3 recorded interview and transcription below.

Give me a little bit of background about your experience with software development and application modernization.

I started as a programmer and then became project manager and then became a freelance developer and got a group of freelancers sort of working for me. Then, back twenty years ago we began developing too, just after Synon had come out that was also a type of development technology, but we soon realized that we were rather eclipsed by the already existing contenders in that. Secondly, quite surprising to us, people really needed help in understanding and re-working their old applications. We hadn't intended to do that, but it sort of got thrust upon us. The tool developed more and more as a re-engineering design recovery tool, which basically extracted all the RPG and COBOL applications into a comprehensive data base. Then, we can automatically rework the software from there. We hadn't anticipated being successful in that area because there were already tools for that, but it turned out they weren't satisfying the demands. We really had a big hit with using that software to form fairly mundane re-engineering for things like euro conversions, field re-sizing and most importantly year 2000 conversions. Using our technology, you can write programs and we did write programs that automatically modernize the software in that mundane sense and since making it actually cope with the year 2000 change or similar change. We can make that process 100% automatic which is what the opponents said could not be done. So, it got to the extent that people were sending their software out to be converted. We could convert it completely, send it back to them the next week and charge them a huge bill. We made an awful lot of money from that which has since the year 2000 just been re-invested in the company.

Having a way to do that automatically sure makes a difference. A lot of companies had to read a lot of lines of code.

Yes, we also used to do that. There was a sub part of the company that did that for 2e applications. Perversely, it wasn't so easy to make that 100% automatic. No defect of the 2e code base, but simply because there is a relatively smaller number of customers for that, so it wasn't possible to alternate quite so efficiently.

As you've gone through the history of your company more recently, what are some of the primary reasons that you see companies needing to modernize their applications?

There are three reasons. The top reason is for productivity. Modernizing and working with an application that consists of millions of lines of 3GL code, Cobal or RPG is just not very efficient.

The second reason is to work with a more modern user interface, though it's possible to have RPG programs work with web applications with CGI, but obviously the language wasn't written for that whereas that type of user interface and that type of infraction is native to the languages like Java, VB, C- Sharp etc… The third reason is to make use of a much larger base of people, programmers and software houses that work with the language. With RPG you've got a rather restricted pool of staff and outsourcing companies that you can deal with. With almost all of the people new to the industry I find it much easier to work with one of the newer languages and forces of competition being what they are, you get a much better deal from an outsourcing company.

In order to modernize there are several different approaches that are out on the market right now. From simply changing the screens to completely re-writing and re-engineering the application. What are some of the other kinds of approaches that you see?

Well, let's do it at a scale of strategic options that you have. As you mentioned you can simply opt to put a new sort of screen on the front of the old application. It's a lipstick on a pig kind of strategy. I would argue that you really haven't modernized then. You've actually just put a different sort of terminal on the front, so it's really an un-modernized application. A second strategy you can have is to essentially convert it where the code will effectively stay the same. Line for line converted into a new language. It's difficult to see the point of that because you won't actually get any better productivity; you'll actually get worse productivity because the resulting application will be difficult to work with whether you're an expert in the new technology or the old technology. New programmers would look at it and think "I would have never coded it like that." It's analygis of system 36 conversions they've done in the past

Is that where you get a language sort of like Jobal? You know when you get a combination of Cobal and Java mixed together.

Yes, well basically what you're going to get in a stait language conversion, the RPG or Cobal instructions will have been converted into say the Java instructions. There usually will be more lines of Java than there were of RPG , simply because some of those instructions do more things and you have to emulate all the specific features in the old language in the new environment. You will end up with Java that nobody could possibly have written. You have a system that's totally peculiar to a converted system. The whole point and productivity of the new language comes from that which is written in an object oriented way. If not, then it loses any advantage at all. So, I would argue there's basically no point in that. The third and fourth alternative, the third alternative is to re-write it without tools, which is going to achieve the goal but it's going to achieve the goal rather slowly because you can only re-write software at a

certain pace. It also tends to confuse the issue as to whether the new application needs re-analyzing from the business point of view.

The fourth approach, which is our approach, is really a variation of re-writing it. It's to re-write it automatically using the recovered design. In other words, to recover the essence of what the old programs did to rebuild it to the same quality standards as you would do as if you were re-writing it from scratch. Then, of course, you've got to make sure that you're retrieving all the peculiar details of the old application.

You actually end up with brand new programs, but using the legacy logic and so forth.

Yes, the Phoenix approach. You're building it again on the ashes of the old.

When you use your approach, which sounds like a very good approach, what does this kind of project consist of?

In the first stage you've got to actually recover the design, which is the application of our core tool X-Analysis and just from the business point of view, from our companies commercial point of view, we have found to our surprise, that a lot of companies get enormous value purely out of that first stage. They don't actually have to go very much further to gain value from the project. For some people the modernization goal is a little way off now the planning can take several years. The second stage is to break up the application into projects areas. We call them application areas. So the modernization can be staged. Basically, our approach because this human intervention is going to take time. You have to decide in what phases you're going to modernize. You have to split up the application from a business point of view. That is also an output of the sub-division of the application that has great value from the user's point of view quite apart from modernization. Then the third phase is to actually apply the modernization automatically. In other words to recover the design and re-build into the new application from all the recovered design constructs. Include the recovered screens the transaction details when the files are read or updated etc..and the business rules. Determining the validation and subsequent processing. That's all done automatically. The fourth stage is the audit process which is to check that in fact all the details of the old application are in the new application and to apply corrections.

When you're modernizing your project is there a certain application architecture that you're trying to move to? Does it really matter what architecture you are going towards?

Well, what we find is that regardless of the new technology that the customer is going to use, the specific language. Whether it's VB, C#, Java, PHP, etc. Really people are aiming for the same type of application architecture broadly. There really is strategic agreement on that. What we mean by that is you're moving from a procedural approach to a screen based event driven approach. In other words where the actions of the user on the screen drive the program, rather than the programs driving the user. It also can be an object oriented approach where the

functionality, instead of being replicated and repeated many times in the Legacy programs, its encapsulated in new objects. I find that consistent.

In your approach, you don't necessarily have to target one specific language. You can actually per project, determine what language you want to build your final programs in?

That's absolutely right. From our point of view the generational language doesn't make a huge amount of difference. We favor the languages that are most similar to Java, but basically that is most of them…so we're talking about Java, C#, PHP, EGL, but it won't make a huge amount of difference if you use VB for example.

In your process you're recycling your code into modern language. It seems like that's a complex process. How are you able to accomplish this?

Fundamentally by design recovery. In other words, when we have a 5000 line RPG program we're not aiming to produce 5000 lines of Java. The design recovery process is first of all finding the screens and mapping them as new screens. In other words, it's producing what we call function definitions. What in 2e systems are also called function definitions. In other words, enhance screens that not just define the look of the screen, but where the data comes from, how it's validated etc. and also how the screen is processed in terms of what are the further programs that are called from there. Within looking at the actual logic that is processed as a result of the command keys or enter key on each screen to see exactly what happened. That can either be a logical action, like a call to another function or it can be a business rule. In other words, conditional processing based on the value of one of the database fields or fields that were entered on the screen. The logic is built a fresh from that design. So you'll end up with a much smaller amount of code. Of course there will be some bits of logic and peculiar methods of screen flow that don't easily translate and that's what the audit process ends the projects on.

Now, you mentioned earlier that most of the older languages are in procedural and you're moving toward object oriented. How do you handle that conversion from procedural? They work so much different from an object oriented approach. How do you accomplish that?

Right…well, the very first primary difference between the procedural languages and the newer languages is that the newer languages are event driven. In other words, instead of the old language actually driving the screen, in VB or Java or any of these languages, the event on the screen drives the logic. Now we have an automated process that we've built up over many years that essentially does this. That actually looks first of all after after having found all the screens, then finds what the screens do. Like turning the whole program inside out so that it becomes a set of procedures that are called, invoked from each screen action, not just written in one long screen of logic. Then the second aspect, in terms of processing driven by each event, we're turning it into business rules. In other words we're identifying the purpose of each particular snippet of logic and rebuilding that as an object.

Then, I guess after that you have to make sure that the programs actually fulfill the business logic. Your audit takes care of this. Is that correct?

Yes, exactly. Actually, I just want to make one further addition in a previous answer. There is an exception to what I just said, which is if you start with a 4GL system like 2e, perversely that's already written in an event driven model so you have an extra advantage there. The Logic always was written in 2e and some 4GLs to event driven models. That makes that quite precise so that you won't actually get. If you're in a 3GL situation with RPG or Cobal, basically there's a final audit stage, which we provide you screens or interactive displays that allow you to see side by side the old logic from the program and where it's been mapped into the new objects and it specially highlights any lines that have been left behind and then you have to decide what to do with them. That is the audit process.

When you're looking at the Legacy system, what are the two most important assets of that system that you look at in the modernization project?

Essentially, we're looking at transaction definitions. In other words what in the 2e system would be called functions. Enhanced screens and what the screens do and where the data comes from and how it's joined together, the join rules. Secondly we're looking at the business rules, which are the process rules. 2e systems, that's action diagrams, but more generally it's the Logic/ Java, logic beans that are driven by events on the screen. We're extracting that automatically by analyzing the RPG or Cobal code.

You're extracting the business logic without programmers reading the code. What is it that you're doing to extract that logic?

Basically, it's processing the code, looking for business rules. Business rule, dictionary definition, is any piece of logic that conditionally depends upon the value of a database field. The first thing we've got to do is to map where every single work field in the program comes from. Which is the same thing we used to do in year 2000 X-Analysis periods. The unique thing about our tool, the basics analysis tools, is that it has a knowledge base of every line of every variable. So it's able to map what every variable means. Then it's looking for pockets of logic that are driven by the conditional value of fields derived from the data base. It then finds what the action does. The action might be a validation or it might be some miscellaneous piece of logic. If it's a validation for example it would look at what is the convention by which that application puts errors on the screen. Perhaps it said a message number or an indicator and it will de-code what that really means in terms of the application and what is the message description and will turn that into a pocket of logic that is described in our re-engineering data base. That will be converted to the new system.

In part of your process you actually develop a data model. How does having a good data model help you in writing code into the modern language?

Basically, because it enhances the productivity of the new approach. Any modern system tends to have the entity relationships explicitly described. We know that for example the orders file refers to the stock file. That information can be used or is used in most development techniques to make sure that the data base accesses automatically know how to join the data and they automatically know how to validate the data, which is quite a productivity boost. Now, the System i is fairly unusual in not having that entity relationship information unless somebody had happened to use the very latest features provided by IBM. What our tool does is it automatically analyzes code to see what entity relationships existed in the legacy code and then it makes that into an explicit data model. Then any new functions built will be re-engineered or whether they're built entirely from scratch, but automatically know about those modeling relationships.

Let's see… DDL, where does that come into converting to a new language?

Converting the System i application , converting it from DDS to DDL is not necessary, but it is quite advantegous. The two advantages it will give you is first of all when you convert the DDL, it will then describe the fields by their long name, the textual names which of course is the way you refer to them in 2e for example. That type of reference is absolutely standard in the newer languages even if the programmers do not expect to be reading pneumonics through the code. There are ways in which you can write new language using hibernate for example. So that you work with new alias names in spite of staying with DDS, The second way DDL helps is it will improve the performance of the application quite considerably. When you move to DDL you move to an SQL defined database. Data is validated in a different way in an SQL defined database. It's validated when you write it, not when you read it. And that will make quite a considerable performance improvement and make it easier to write new SQL based applications and every new language, I think without exception uses SQL to access the database rather than record based IO as the RPG applications do.

SQL is definitely getting to be used quite a bit. I was reading and saw you talk about the application skeleton that's developed out of the data model. What does this consist of and how do you write that?

As we said before in our technology, as with in fact almost all 4GL and model based generators what happens is we take the recovered design and we build a new program using a template program. In other words a core standard design for that type of program that we store in a little data base. We have skeletons, like templates already defined for all the standard types of transactions such as entering data records, display, sub file, transactions, etc. Our technology allows one to feed in more types of skeletons if you encounter very unusual types of legacy logic. Basically it consists of the model programs and the model business logic for any particular target language C#, Java, PHP etc..

Now UML is one of the modern definition languages. Do you use that in re-building the Legacy systems?

Well yes, though UML is not necessary to our approach, but it can be used to modify the recovered design. Let me explain. The recovered design is extracted into our database. Which can then be translated and is translated into UML so that you can see the transaction logic and the entire logically recovered design in UML format and display it in rational tools for example or if it's built in eclipse tools that works out of the open source eclipse project and you can use that UML to decide on any changes you wish to make to the application so you can work with true model based development in your re-engineering stage. In other words modify the application. After its modified the design, after it's been recovered and build from a slightly modified design. However, it's not necessary to do that, if you wish for a quick project you can just avoid the step of looking at it in UML and generate it to the target language straight away.

But if you do use UML, then you're able to maybe correct some of the issues that you've had in your Legacy system.

Absolutely, but a typical issue would be that the Legacy system has been built around the constraints of the 24 x 80, 5250 design. So in other words you can't easily fit many fields on the screen. So the application was splitting to a number of transaction screens artificially in order that we didn't compromise that screen size limitation. In the new application, you've got much broader constraints there. So at the UML level, you can choose to merge different screens together.

Say you've done your analysis and you've seen now your Legacy system has some core issues, maybe in the database design. When would you address that in your modernization project?

You would do that before going to stage three. In other words, before doing the generation of the new application. You would generally be more efficient in other words to perform your modifications while you were still in the Legacy mode. So you should aim to correct the final definitions while they're still in the X-Analysis model of the application. You can of course correct those in UML, but then you'd be doing things by hand. The beauty about making changes in the X-Analysis model is that one can do the work programmatically just as we used to do for the year 2000. In other words you can make global changes and this is quite often what our tool is used for even when the customer is not considering actual modernization at the moment. They simply wish to accommodate the change to the database design or an increase in field size or change of keys etc..can all be done automatically.

That's a pretty nice ability to do; because a lot of times in a modernization project you do have some glaring issues that you want to address. It sounds like you could potentially address those here while you're modernizing. So you end up with a much better product in the end.

Absolutely, and of course another factor which I didn't mention is that your existing staff are much more familiar with the current technology than they are with the new technology ..so it reduces risk to make those type of modifications while we're still in Legacy mode so to speak.

So now, let's move on a little bit in the phase. We're ready to generate our actual programs. What process do you follow to generate the programs? Is it all done automatically that you generate them or do people have to look at the logic that's been out there in write programs?

They don't need to write programs. What does at the generation stage, there is scope for manual intervention. Basically we are a logic viewer. My logic viewer presents you with a screen of all the recovered design, of all the recovered business logic with the blocks of business rule logic that are going to be recovered or highlighted on the parts that are not recoverable or not highlighted . It then allows the user, the programmer to work through that and change the design decisions made by the product. Of course if you find that the product is making too many wrong decisions you would then confer with Databorough about changing the parameters of the tools so that the conversion is more accurate.

Once the application then is written your audit step comes into play. How do you audit the generated programs to make sure they're doing what they were supposed to do in the beginning?

Well, there are two stages to deal with it. The first stage is to audit what Legacy code was carried across and what was ignored. We give you special user interface for analyzing that. The product itself highlights exactly which logic was converted and where it was placed in the new VB or Java program. You can then look at your legacy logic and see which pieces would have been left out. That's one thing. That's stage one and that may of course require some extra manual work. Hopefully, not too much, but there is certainly, in a more complex software, a need for that. Second stage you're actually auditing that it actually works and that generated form will have to be done by testing. There's no way around it. This is simply an enhanced, greatly enhanced method of building new software, but as with all software you've got to test it.

So right now you don't have an automated testing process? Do people just have to run though the testing themselves?

We don't supply automated testing tools, although there are other vendors that do and we can put you in touch with them.

So you do end up with a good product that you can work with and test at that point?

Absolutely, that's the whole point.

If you could compare this process to someone going in and manually doing all these steps… what kind of time are they going to be saving?

We will save approximately two thirds of project time. So if it would have taken you six months manual it would take you two months with automation is the rough rule of thumb. Of course sometimes people under estimate how long it would have taken manually, but we think that's quite accurate.

Are there any shortcuts that someone could take along the way or do they pretty much need to follow every step all the way though?

Well, the major way you can make a much bigger time saving in that two thirds is by piloting and prototyping the approach so that you increase the scope of the automation. This would often require us to be helping with the project by improving for example the application skeletons way more suited to the way that particular legacy application worked. By repeated work with a prototype application over quite a number of weeks, maybe months, one can streamline the automation approach. It sort of becomes 90% automated with very little manual intervention.

90%...that's a pretty good number, especially for a real large application.

Well, absolutely….you have a choice and this would come out in the application sub-division. Whether you wish to do a small piece of the application at a time or whether you're aiming to convert the complete software houses you might well choose to do so if they've got a complete application and the new technology. If you're aiming for the latter then it is probably worthwhile to buy in some help from Databorough to streamline the modernization technology so it really suits your application and you can get the whole thing done really quickly after that screen lining process has taken place. Of course during the screen lining process it may take a couple of months but your own staff don't actually have to spend much effort in that period. A special case of course is with existing applications that already were model based. In other words they were built from a 2e model or some similar technology and were developed by a particular technique even just using RPG code. In those cases if we can guarantee that all the software fits a certain model which is a great advantage to those approaches that you know what to expect. Then it may well be possible right now to achieve that sort of 89%.

A model based language like 2e actually lends itself towards being modernized more quickly than RPG or Cobal application

Absolutely..the 2e application, to take the extreme case was already written in what is effectively a modern architecture and that performed in which it is in the model. The only trouble is that as soon as it's generated to RPG it's in really a very old legacy structure. So it's actually perfectly primed to be generated as Java or similar.

A modernization project is risky and expensive. How can companies address the risk and reduce the risk in a modernization project and maybe come out in a more cost effective process?

Essentially, by doing the modernization piece by piece is one thing .The second is by deploying much more technology too, rather than deploying many of their staff on it. The great risk with modernization projects is that staff spend a lot of time writing software and then abandoned because there's a change in approach decided on later and this happens quite frequently and the modernization takes so long that a newer technology comes along or the current approach is proved not so useful. The beauty of doing the modernization with a tool with a high degree of

automation is that if a change of technological strategies is decided upon and there are quite a few possibilities there. Different UI Technologies, user interface technologies, different development languages quite a lot of super Java technologies like Websphere Smash as an example. Ruby, Grails, Groovy are new technologies, then cost of adapting to those new technologies comes down to the tool provider. I asked that actually to the company performing the conversion and so that reduces the risk a lot.

This process, can it be effective for large shops and small shops? You know small shops maybe have 2 – 5 or 10 programmers that are busy doing their work and then your big shops have maybe hundreds of programmers.

Well, yes you can use the modernization approach on either end, but they will use it differently. Where a large system would be relatively much greater stress on the analysis stage, so the generation approach becomes more of a downstream activity. With a large shop it's much more critical to know that you're re-building the right portion of the system, you're building it in the right way. It's a system analysis and the use of the recovered design just to establish exactly what you're trying to do and it links with the part of the soft ware that's staying in the legacy structure become much more critical. With a small shop you're laying for a much more higher degree of automation to achieve a more straight forward objective.

You mentioned a little bit about the kind of assistance you provide. Can you elaborate a little bit more on the kind of assistance Databororough does provide for customers?

Essentially, in the first instance of course we've got to train you in the use of the tool and the use of the approach. Secondly and perhaps most importantly, we've got to prove that the approach will that the approach will produce exactly the end result that you need and it's much more efficient than if that's our cost rather than the customer's cost. So essentially we will take a portion of their software and modernize it or alternatively we will undertake just streamlining the approach with new skeletons and new techniques to achieve their precise objective. Basically, rather than asking the customer to experiment with the applications and tool to their particular legacy techniques, we'll do the early stages of the project for them and then they'll take it over.

Then the tools that you provide. Do you market them separately or are they a package of tools that you use to modernize? How does that work?

We supply our tools in a module based form. The modules are cumulative. In other words you always start with the basic X-Analysis which I always think is now the industry standard documentation cross referencing tool. Then you move to re-engineering and data modeling. Then you move to design recovery and you basically decide to buy more or less of the tool set. Or you can start small and just buy the add on models which essentially doesn't cost you much more.

…And if someone's interested in finding out more about your process and your tools and your customer support. What's the best way for them to reach you?

Just log on to www.databorough.com . That's databorough.com. and you can automatically download a free trial of the software. There's all sorts of materials explaining how it works. You can test all of that out remotely. As soon as you're ready to talk about it, send us an email.