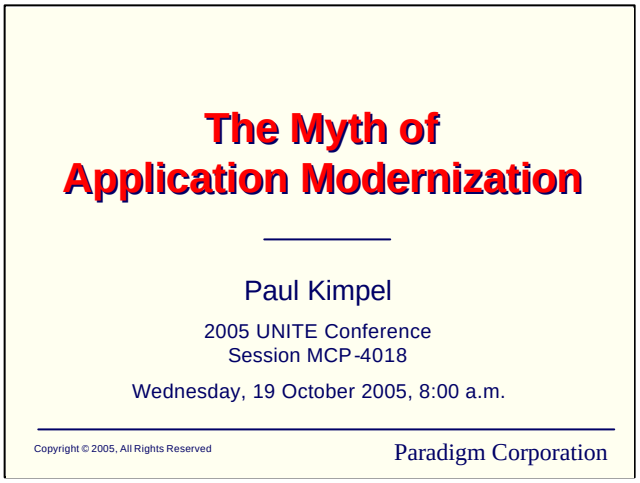


The Myth of

Application Modernization

2005 UNITE Conference
Minneapolis, Minnesota

Session MCP-4018

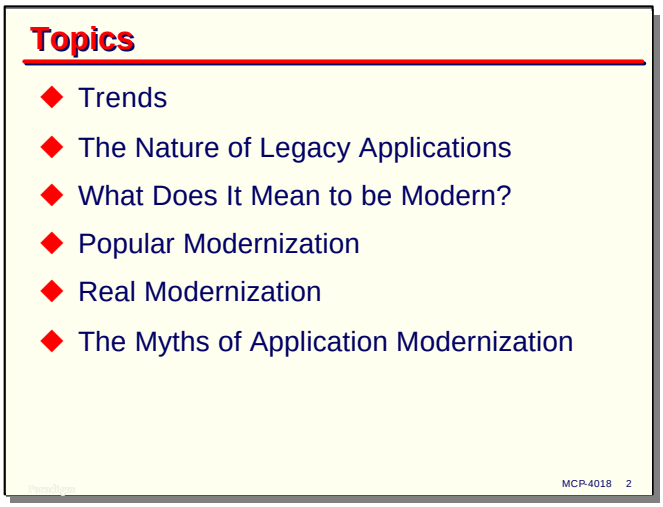
Wednesday, 19 October 2005, 8:00 a.m.

Paul Kimpel

Paradigm Corporation
Poway, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



I have been thinking quite a bit over the past several years about the state of our current mainframe-based business applications, where software development technology appears to be heading, and what that means for the future. At the same time, I have become very dissatisfied with the state that most legacy applications are in, and what I hear of attempts to move them into the future.

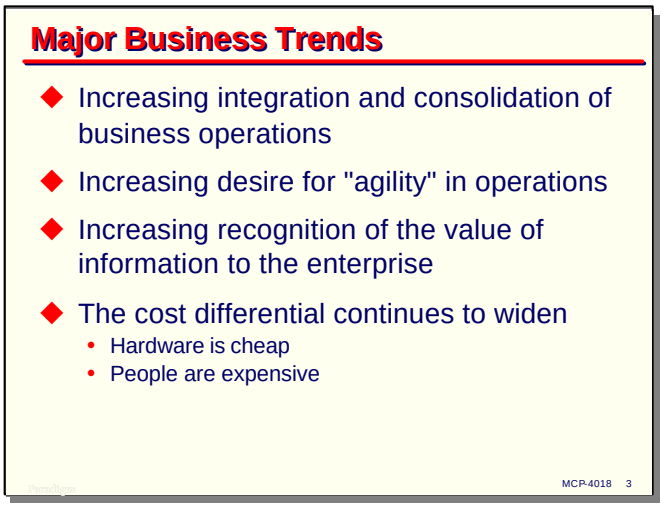
This presentation is an outcome of that thinking. I'm a solutions person, and I wish I could stand up here and say that I have a solution to how our enormous collection of legacy applications should be carried forward into the future. I cannot. But then, if I was that good, I wouldn't even be here.

As a result, this talk is more along the lines of a progress report on my thinking. I hope that these thoughts will stimulate your thinking, and perhaps some open discussion on how we, as builders and maintainers of business applications, and Unisys, as a major technology provider, should be facing the issues surrounding our aging, but vitally important, application software base.

There is an idea, backed up by a buzz word, that our current applications need to be "modernized." This is the basis of my presentation. We can talk about this idea in a couple of ways. There is "**A**pplication **M**odernization," with big capital letters -- for which a number of enterprises are developing products and services. There is also "application modernization," with lower-case letters, which is concerned with the concept and possible approaches we might take towards moving our software base into the future. It's this latter sense of application modernization that I am really trying to address. So, as the lawyers might say, the presence of capital letters in "Application" and "Modernization" in these slides and notes is purely a matter of typographical convention and not intended to reflect on any particular product or service that may be offered now or in the future.

I'll start by briefly discussing what I see are a few major trends in business, especially as it intersects business IT. Then I will talk about the nature of legacy applications and my take on how we got to where we are today. As the final part of setting the stage for my main points, I'll discuss something about what it means to be modern.

With that as background, I'll present a currently-popular activity that is often associated with application modernization. I'll discuss what I think are the benefits and detriments to that activity. Since I think that activity has some detriments, I'll then talk about what I think real modernization entails, and my ideas on how to go about it. I'll conclude with a summary of what I perceive as the myths of application modernization.



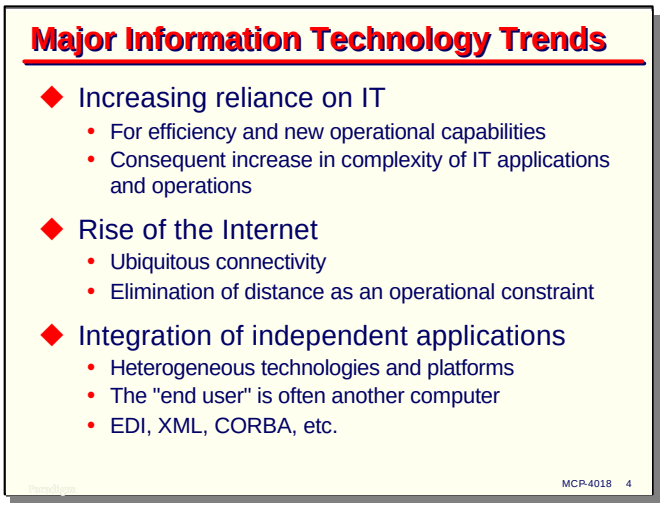
There are a few business trends that I see having a major impact on IT, and especially on the legacy suites of applications that have traditionally been at the heart of enterprise operations.

The first of these is the increasing integration and consolidation of business operations. Just about everything in business operations is connected in some form to everything else. A large amount of the overhead in an organization comes from managing those interconnections and the information that flows along them. The trend is to automate more and more of that management and information flow so that people don't have to do it.

A second trend, which is receiving a lot of attention lately, is that of the increasing need (or at least desire) for more "agility" in business operations. Business is always changing, and often the ability to adapt rapidly to changes in the market can yield considerable competitive advantage. Alas, implementing such changes often requires significant modifications to operational systems – both manual and automated and simply understanding what the impact of the changes are, let alone implementing them, is often a major drag on adaptability.

A third trend is the increasing recognition, at all levels of the enterprise, of the value of information. Partly this comes from the desire for agility, but also from the long-held belief that the better your information and analysis of it, the better the decisions your enterprise can make. It's also true that every business, in whatever sector of the economy, is becoming more and more service oriented. Service businesses tend to be almost entirely information driven.

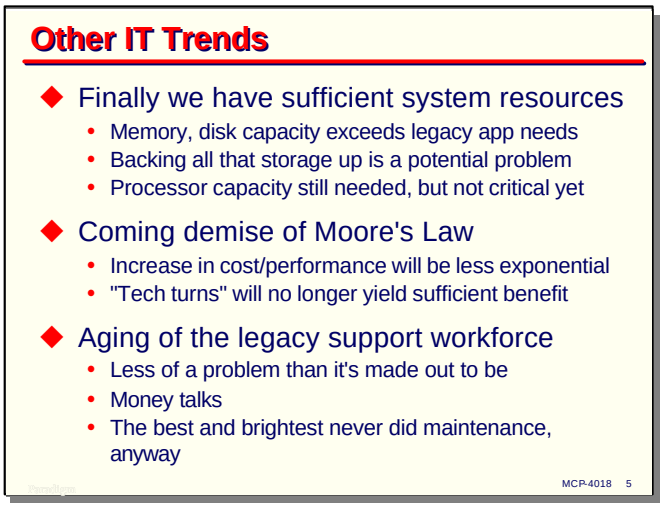
Finally, there is a powerful cost differential that continues to drive automation and software deeper and deeper into the enterprise -- computers are cheap and getting cheaper, while people are expensive and getting more so.



Along with these business trends, there are some major trends in information technology. First, because of the increasing degree of integration in operations, desire for agility, recognition of the value of information, and increasing hardware-versus-people cost differential, enterprises are looking even more to their IT infrastructure for solutions than they have in the past. This has always been the *raison d'être* for IT, but in the last 10 years is has become much more pronounced.

The second IT trend, which has become so much a part of our lives that we tend to forget how truly significant it is, is the rise of the Internet. The World Wide Web is certainly a major part of this, but to me the real significance of the Internet is that it has given most of the world (the first and second worlds, at least) the prospect of really cheap, really reliable, ubiquitous connectivity for data interchange. What the Internet has effectively done is eliminate distance and physical separation of parties as a constraint for many business operations – it no longer matters where the computer is or where the user is. Compared to what we were able to do in this area just ten years ago, what we can do today is amazing, and we have barely scratched the surface of what is possible.

The third major trend that I see is the integration of what were once independent applications. It was not so long ago that most IT organizations developed all of their own stuff. They may have purchased operating systems and utilities from third parties, but the apps were home grown. That has now changed quite a bit. IT shops are now being called on to integrate the functionality of applications that were developed independently, often using entirely different technologies, and in many cases, residing on completely different types of platforms, perhaps communicating over the internet. As a result, the end user is in some cases no longer a person, it's another computer system. The consequence of this is that interfaces have to be much more carefully designed, since the other end of the dialog is no longer able to react very creatively. There has been a great deal of interest in standardizing and generalizing these types of interfaces, which has led to a lot of work in areas such as EDI, XML, CORBA, along with the rest of the alphabet soup of business-to-business technology.

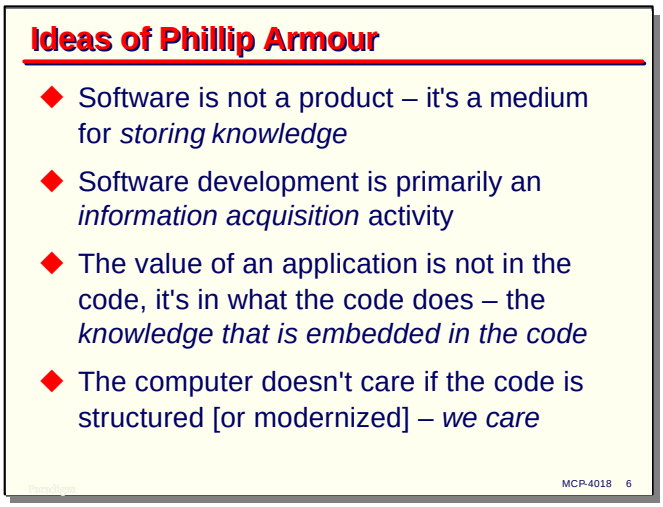


There are a few non-business oriented trends in IT that I also think are worth mentioning.

First, our computer systems finally have sufficient resources to support what we have been trying to do with them for the past 40 years. There's probably no such thing as too much resource, since every cost-effective increase in physical resources merely opens the opportunity for us to attempt things we never could before. In general, though, most of us have more memory and disk capacity available to us than we need to support our legacy applications. If there is a current limiting resource, it may be in processor performance (or perhaps in the disconnect between the performance of processors and memories), but I don't see that as critical just yet.

The second trend, which I am just beginning to see on the horizon, is the demise of Moore's Law. Performance will continue to increase, to be sure, and cost/performance will continue to decrease, but I think we are going to start seeing that growth curve droop. If that is the case, it's going to have a major impact on both the software industry and organizations with growing workloads. It's going to mean that increases in performance and throughput are not going to be as realizable by simply doing a "tech turn" on the equipment. We are going to have to start looking at the design and implementation of our software systems to achieve greater performance. We should have been doing this all along, but the ever-increasing hardware performance and ever-decreasing cost/performance ratio have made it unnecessary. I believe the free ride for improved application performance is coming to an end.

The third trend is getting a lot of press lately. It is certainly true that the two generations who have built and maintained the bulk of our software applications are getting older and are dropping from the workforce. I think that this trend is a lot less of a problem than it's often made out to be, however. In the first place, money talks, and people respond to real economic incentives. If the world needs more COBOL programmers, then the price for COBOL programmers will simply rise until enough of the younger crowd sets their distaste aside and decides it's in their interest to learn COBOL and work on older applications. It's not like it's really hard, after all. In the second place, the best and brightest in this business, whether they do COBOL programming or not, never did maintenance. All of those average Java and VB programmers out there will make perfectly adequate average COBOL maintenance programmers, which is where the shortage will really occur.



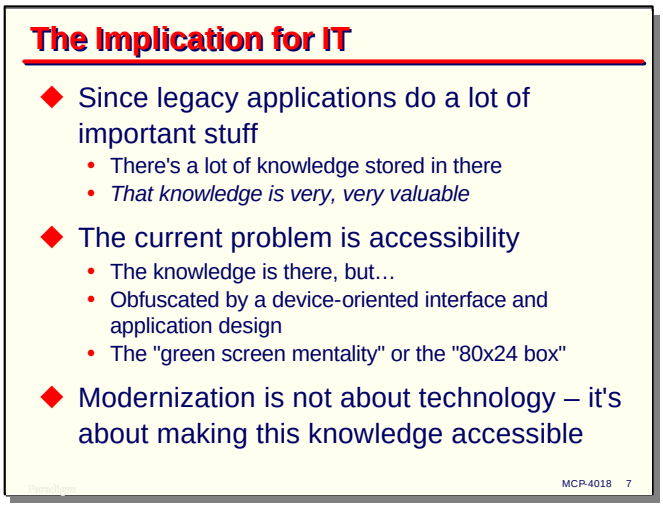
There is another aspect to current trends that I want to add to this mix, namely a changing appreciation for what software is. I've discovered a series of articles by a consultant named Phillip G. Armour, who writes an occasional column in the *Communications of the ACM* titled "The Business of Software."

Armour's thesis is that software is not a product. It is instead a medium – in particular a medium for storing knowledge. There are five such media: DNA, brains, hardware, books, and now software.

He goes on from this to propose that, since software is a medium for storing knowledge, software development is really not about generating code, it's primarily an information acquisition activity – the discovery of knowledge. As anyone who has developed a non-trivial application has discovered (often to their extreme discomfort) it's what you *don't know* about the problem that causes all of the grief.

Therefore, the value of an application is not the application's code, it's in what that code does (and all too often in what we have learned, the hard way, it shouldn't do). In other words, the value is really the knowledge that is embedded in that code.

Furthermore, Armour says that all of our posturing about whether the code should be this way or that – structured, modernized, object-oriented, whatever – is entirely an issue for us, the developers and maintainers. The computer system does not care whether the code is structured, or modernized, or however we think it should be. The hardware just follows our orders. It's *we* who care. So when we talk about modernization of legacy applications, we're talking about what's in it for us.

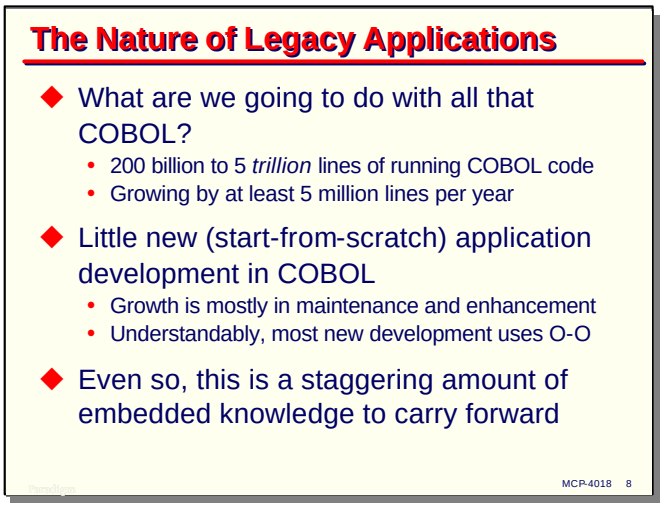


The implication of these ideas for IT is straightforward. Since our legacy applications do a lot of important things (if they didn't, we wouldn't keep them around), there is a lot of knowledge stored in them. That knowledge is very, very valuable.

It is also the case that this embedded knowledge is fairly inaccessible, except within the context of the legacy application environment itself. The functionality is there, but in most cases it's hidden behind device-oriented interfaces (80x24 green-screen forms, 132-column printed reports, and the like). The knowledge is all mixed together with, and obfuscated by, the grubby details of accessing it using the technology which was available at the time the applications were built. The knowledge is further hidden by application architectures that are oriented solely to that old technology. Most COMS transaction processing (TP) programs are concerned a lot more about COMS and screens and DMSII than they are about the knowledge that is embedded within them. I call this obfuscation the "green-screen mentality" or the "80x24 box."

Therefore, application modernization is not at all about technology, although technology will certainly have a role in achieving modernization. Modernization is really about making this embedded knowledge more accessible.

This is why the wholesale rip and replace approach to modernization has not fared well to date. Replacing COBOL/mainframe apps with Java/blade-farm apps is a technology transfer. Transferring technology is not the point. The point should be transferring knowledge from one implementation of an application to another. What a number of early attempts to do this have found is that it's quite difficult to recognize the knowledge embedded in the old code, let alone extract and transfer it successfully.

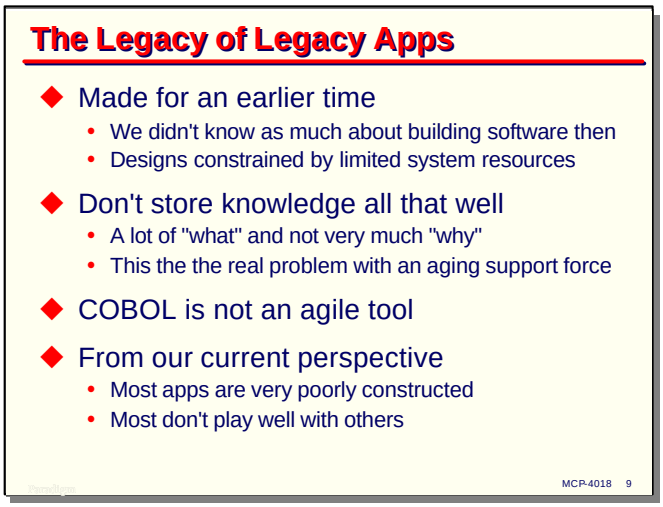


With that background, let us turn to the general nature of that which we want to modernize, the legacy applications.

What are we going to do with all that COBOL? Estimates by the Gartner Group at the completion of the Y2K remediation effort say that there are between 200 billion and five *trillion* lines of running COBOL code in the world.[1] Gartner estimated that body of code was growing at five million new lines per year, but I think that number is low, and is probably more on the order of 100 million. I believe that most of this growth is in maintenance and enhancement of existing systems, though, and not in the development of new applications. Quite understandably, most developers want to base entirely new development on object-oriented technology.

No matter which end of these estimated ranges you believe, the amount of detailed knowledge embedded in all that COBOL staggers the imagination. It would be a monumental effort just to mine this collection of code for its embedded knowledge, let alone think about replacing it. Such an effort will probably require many generations of effort, if it is undertaken at all.

[1] Ronald Q. Smith, "The Future of Operating Systems", 2002 UUA Conference, Session 1.09, Lausanne, Switzerland, 14 May 2002.



Our legacy applications have left us a legacy of their own. We need to keep in mind that they were built for an earlier time. We did not know as much about building software then. We still have a long way to go in understanding this, but we know a lot more about it now. Most of those applications also were designed under constraints imposed by much more limited system resources – especially small memories and mass storage subsystems. I realized a few years ago that much of what I had learned about building applications isn't important any more, largely because memory and storage capacities have grown such that the old design tradeoffs simply no longer apply.

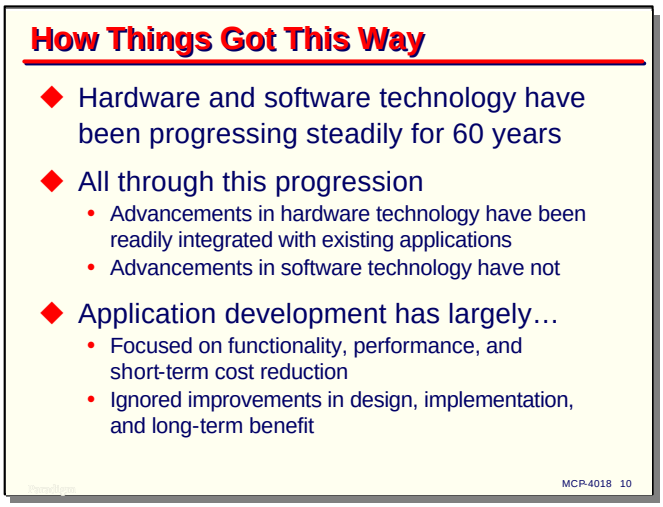
Another issue with legacy applications (and perhaps more modern ones as well) is that while these store a lot of knowledge, they don't store it all that well. Something gets lost in the implementation – much like the loss that occurs when compiling source code to object code. Applications tend to store a whole lot of "what" and not very much "why." This is the real problem with an aging support workforce – they are the ones who know the "why."[1] It is all well and good to complain that this would not be the case with proper documentation, but it is also the case that the code has always been, and will always be, the only truly comprehensive and reliable form of documentation. We need to do a better job of storing the "why" along with the "what."

A third issue is that COBOL is simply not a good tool with which to build agile systems. It is not all that bad for writing individual programs that perform some business function. It's lousy, however, for writing whole *systems* of programs and trying to maintain those systems over time. This is true of all third-generation and earlier languages, not just COBOL. More recent object-oriented languages are much better at this, and some new ideas, such as "aspects," promise still better support for large software systems, but the tool set still has a way to go.

From our current software development perspective, most legacy applications are poorly constructed. They are hard to maintain, hard to enhance, and generally do not play well with other applications. This is a consequence of their time. Perhaps many could have been built to better support our current needs and desires, but they weren't. We have to deal with that.

---

[1] David Grubb, private communication.

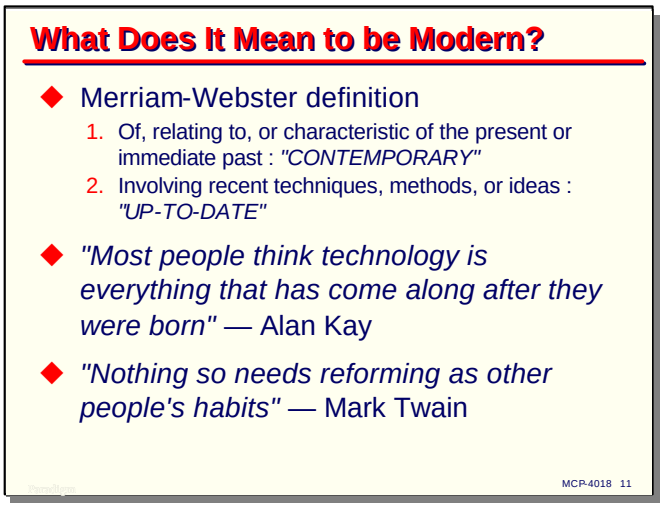


Hardware and software technology have been progressing steadily since the first computers were built almost 60 years ago.

All through this progression, advancements in hardware technology have been readily integrated with existing applications and made feasible entirely new ones. Advancements in software technology, on the other hand have been integrated into existing applications much more slowly, much less frequently, and much less successfully. Partly this is due to the fact that processor architecture has, for better or worse, essentially gone nowhere since the late 1960s, and thus has had very little impact on existing bodies of code. It is also due to the use of relatively standard programming languages, such as COBOL.

Integrating new software technology (and software development practices) has proven to be much more difficult than integrating new hardware technology. Not only does it normally require recoding the entire software base, it often requires completely rethinking the application, how it should be structured, and how the various pieces fit together. Nowhere is this more evident than in the difference between traditional COBOL development and object-oriented development.

Application development (and development methodology) has largely focused on functionality, performance, and short-term cost reduction at the expense of improvements in design and implementation of our applications and the long-term benefits we can expect from them.



What does it mean to be "modern?" There are a few different ways of looking at this.

First, we can use the tried and true approach of looking the word up in the dictionary. My Merriam-Webster Collegiate version shows two meanings relevant to this discussion:

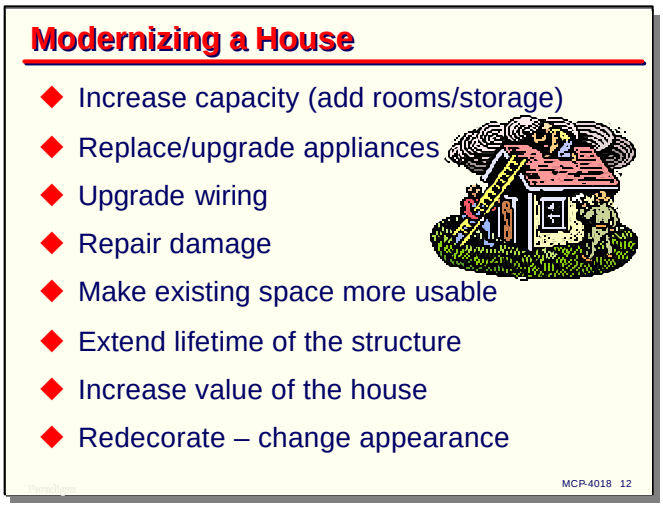
- "Of, relating to, or characteristic of the present or immediate past." In other words, modern means *contemporary*.
- "Involving recent techniques, methods, or ideas." In other words, modern means *up-to-date*.

I suggest that we should be thinking about modernization more in the second sense rather than the first. Contemporary carries the connotation of being in style or acceptable to current tastes. Much of the criticism of legacy applications, COBOL coding, and of mainframes in general falls into this sense of being modern. Being up-to-date means not that newer is better, but newer has the benefit of more thought, development, and experience behind it.

Another aspect of this is that people tend to judge the present based only on what they are familiar with of the past. We also tend to forget that we all stand on the shoulders of our predecessors. Alan Kay put this very well in a talk at the 1986 OOPSLA conference, "Most people think technology is everything that has come along after they were born." In fact, technology is nothing more than the way humans attempt to deal with their environment. Technology is a progression, and as such, it's still in progress. Being modern is simply being at the latest point along that progression.

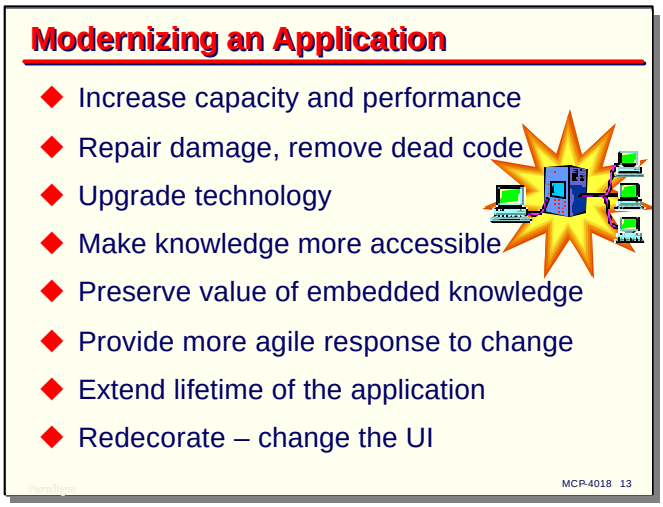
Finally, being modern is a form of religion. Feeling modern is very much a case of feeling superior. This brings to mind Mark Twain's famous saying, "Nothing so needs reforming as other people's habits," which in the context of this discussion could be restated as "Nothing so needs modernizing as other people's legacy mainframe applications."

Once again, the idea of modernity I want to bring forward in this discussion is one of being up-to-date – of taking the lessons we've learned and applying them for our ultimate benefit.



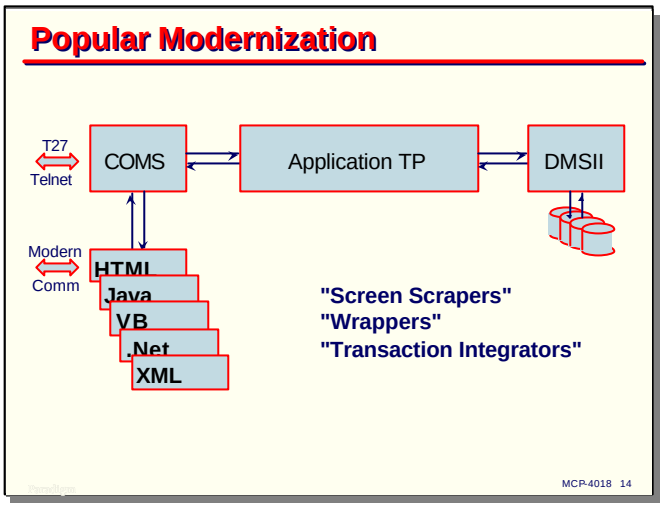
When thinking about modernizing application systems, it's helpful to have an analogy. If we have an older house, we might be considering "modernizing" it. What does this mean, exactly? Well, it can mean a number of things.

- We might want to increase the living capacity of the house, perhaps by adding rooms or expanding space for storage.
- We might want to replace or upgrade some or all of the appliances.
- We might want to upgrade the electrical wiring, perhaps by increasing the number of outlets or their amperage, or perhaps by adding category-5 or coax cabling.
- We might want to repair damage, perhaps caused by water leakage or pests.
- We might want to make the existing space more usable, perhaps by moving walls, adding doorways, or rearranging traffic flow.
- We might want to preserve our investment in the structure and extend its lifetime. This would probably involve applying a combination of the other items on this list.
- Our goal might also be to increase the value of the house.
- Finally, and intentionally last on the list, we may simply want the house look different, or more appealing in some way.



Modernizing a business application has some analogies to modernizing a house. There are a number of ways this can be approached, in various combinations.

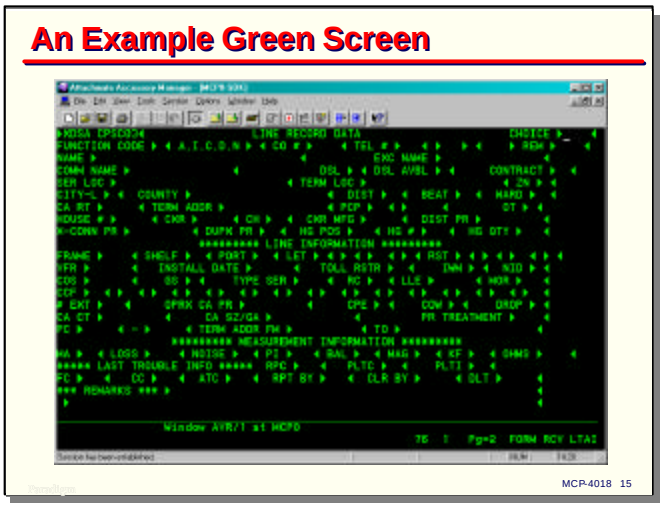
- We could be interested in increasing the capacity or performance of the system. That could be in terms of how much data it can store, the variety of functions it performs, or its transaction rate or throughput capacities.
- We could be interested in the equivalent of repairing damage to a house. Software does not degrade as do physical structures, but it tends to become corrupted over time by changes to the application. There could be dead code that should be removed, or perhaps just bad code that needs to be rewritten.
- We could be interested in upgrading the technology used. That could involve a whole range of effort, from converting COBOL-74 to COBOL-85, all the way to complete re-implementation, perhaps in another language.
- We may want to make existing knowledge embedded in the application more accessible. Perhaps certain types of transactions are possible only from certain types of terminals, or through certain types of communication channels. We may wish to generalize the interface to these so that other pathways can be enabled, in particular the Web and e-commerce media such as EDI or XML.
- We may be interested in preserving the value of that embedded knowledge so we do not have to go out and rediscover it all over again.
- We may wish to make the application more adaptive, or "agile," in response to the need for change.
- We may wish to protect our investment in the application and extend its lifetime. As with extending the lifetime of a house, this would probably involve applying a number of items in this list.
- And finally, as with a house, we may simply want to make it look nicer. That usually involves changing the user interface (UI) to something that our user community will find more appealing.



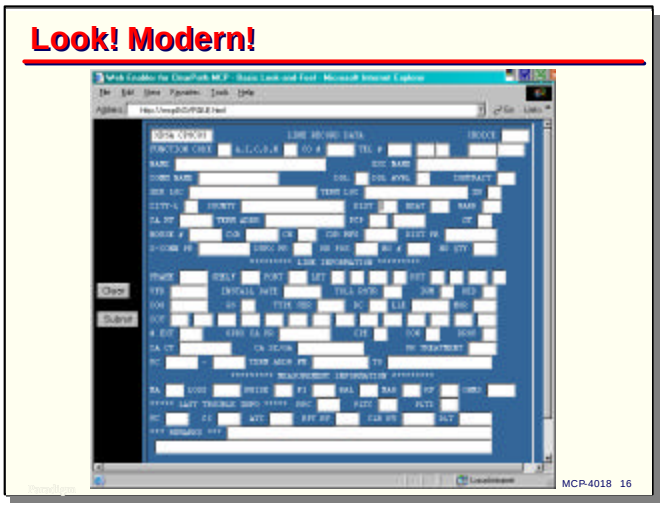
This discussion of what it means to be modern leads me what I think of as "popular" modernization. It's quite straightforward – you simply take the existing application, which is probably built to work with legacy terminal interfaces, and implement a front end system that talks to COMS. This front end accepts the legacy terminal data stream at its interface to COMS, and on the other end reformats it to employ some more modern UI technology.

There are now a large and growing number of ways that this can be done, using an almost as large and growing number of interface technologies from Unisys and others. The new UI can be based on HTML, Java, classic Microsoft Visual Basic, or Microsoft .Net, just to name a few. The COMS interface can be through Telnet, COMTI, CCF, JCA, DTP, or a number of other technologies.

Implementations using these approaches are generally characterized as "screen scrapers," "wrappers," or "transaction integrators." They are popular because they typically require no change at all to the original application.

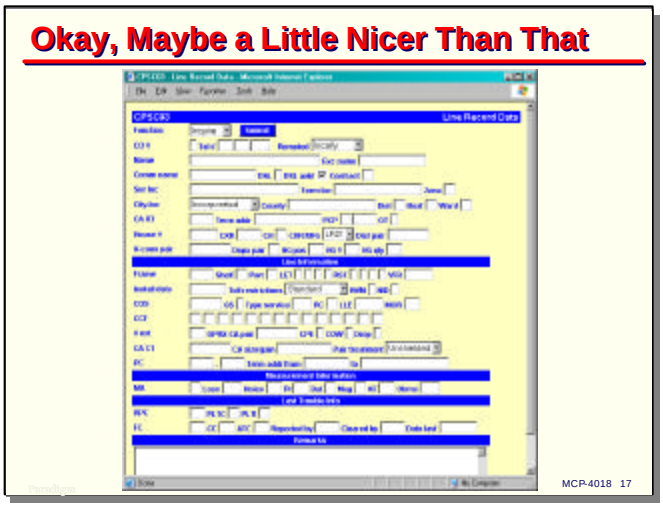


To illustrate the idea of a UI front end, I went looking for the ugliest green-screen form I could find. This is the best (or rather, worst) I could do. Now let's modernize this.



Look! Text boxes! Buttons! A tasteful shade of blue – how modern!

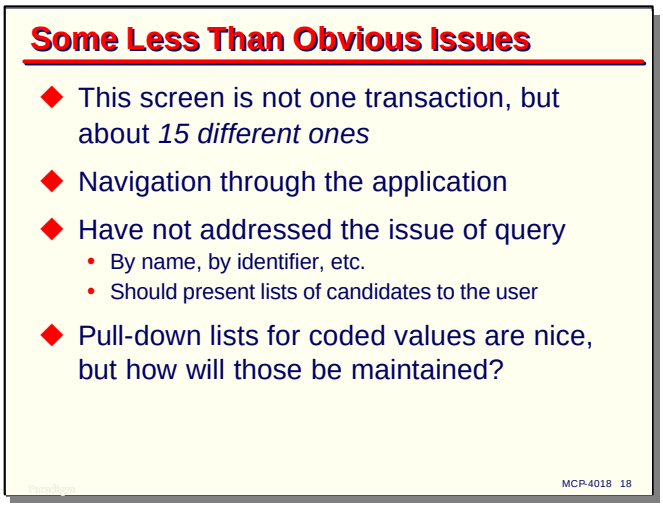
Yes, but it's modern in the first sense of the word we discussed earlier – contemporary, or in style. It's modern in that it looks modern. Actually, unless the user is allergic to green on black, this change doesn't do much for them.



Well, all right, that was intentionally a minimum effort. With just a little bit of work, though, it's not difficult to work up something else – perhaps an HTML presentation that looks nicer and more organized, and hopefully will be clearer and easier to use. In addition to rearranging the fields and cleaning up the layout, I replaced some of the field with pull-down lists and checkboxes.

I copied the original screen layout from a customer's application and don't actually know much about how it's used, so it's likely that someone more familiar with the business issues this screen is trying to address could come up with a more appealing and functional arrangement. Perhaps HTML is not the appropriate choice for this particular application, either.

The point is, it's not that difficult to make the legacy UI look nice.



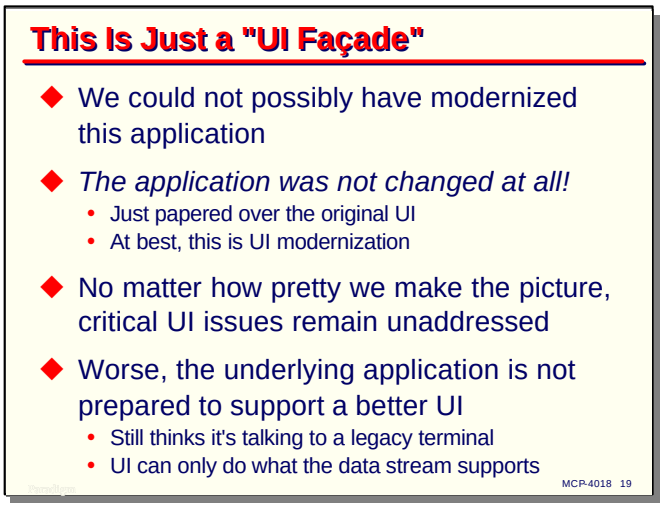
This is really a case of beauty being only skin deep, however. One thing that may not be obvious about the screen is that it's not one transaction, it's used for about 15 of them. Various fields are used for various functions, and you just have to know which are significant in each case.

There are some other user interface considerations that have not been addressed. Navigation is one – how does the user get to this screen/page/form, and how do they get from there to the related activities they may need to perform?

Another issue is finding the data to display to the user, which I'll term "query." Instead of requiring the user to enter identifiers, a good user interface tries to present the user with lists of candidates the user can choose from. This particular screen offers a way to look up an account by name and to page through duplicates, one at a time, but that is slow, and there's no way to back up or skip around the collection of "hits."

I replaced a few of the text boxes with checkboxes and pull-down lists. In the case of the lists, how are they loaded? Are these entries statically defined in the UI front end? Are they values which can be pulled from the data base? Are they dependent on other state, such as which account is being displayed? How will these lists be maintained as the application evolves? These are issues which must be addressed.

Thus, being pretty may look modern, but it may not be better, and considering the maintenance issues we may have introduced, may actually be worse over the long term.



I call this type of change a "UI Façade." That is, it's just a face.

This is not really application modernization. We could not have possibly modernized the application, *because we didn't change it at all.* In fact, we have not even replaced the original UI. It's still there, even if there are no longer legacy terminal interfaces using it. At best, this is UI modernization, not application modernization.

Even as UI modernization, this approach can leave a lot to be desired. It's difficult to implement many important UI principles with just a façade, in particular intra-application navigation, choice lists for lookups, dynamic loading of pull-downs, and sensitivity of the various interface elements to specific user state. Addressing these usually requires bringing significant portions of the business rule logic into the UI front end, or bypassing the application and doing direct access to the data base (perhaps using ODBC or OLE DB), or both. This adds complexity to the overall application and can really drive up the cost of maintenance and enhancement.

What is worse, the underlying application is not prepared to support a better UI, even if we were to design one. While it would be fairly easy to add support for queries and the loading of pull-down lists, what most legacy UIs need is a complete reorganization, decomposing the existing 80x24 screen-oriented user dialogs to something that is more functionally oriented.

Before I go too far in trashing this form of modernization, I must point out that implementing just a UI façade is not necessarily all that bad.

Usually, UI facades are relatively fast, easy, and cheap to do. The result can look really impressive, especially to someone who is accustomed to a traditional green-screen interface. The GUI orientation may make the interface more approachable for new users (but less so for experienced ones, especially in high-volume data entry applications), and if done properly, may even be easier to use. The original interface for legacy terminals can coexist with the new one, since we did nothing to the application that affects that.

Perhaps most significantly, this approach can often yield a short path to an initial web presence. This is especially true for query-only applications, as some front ends are programmable enough that they can perform multiple queries on the legacy end and present the end user with a consolidated and reorganized result.

Whatever the long-term advantages and disadvantages, implementing a UI façade is not a bad way to start on the road to modernizing your applications. You'll learn a lot, and this can be productively plowed into your next phase of modernization.

On the other hand, implementing just a façade does not have a lot of long-term benefit.

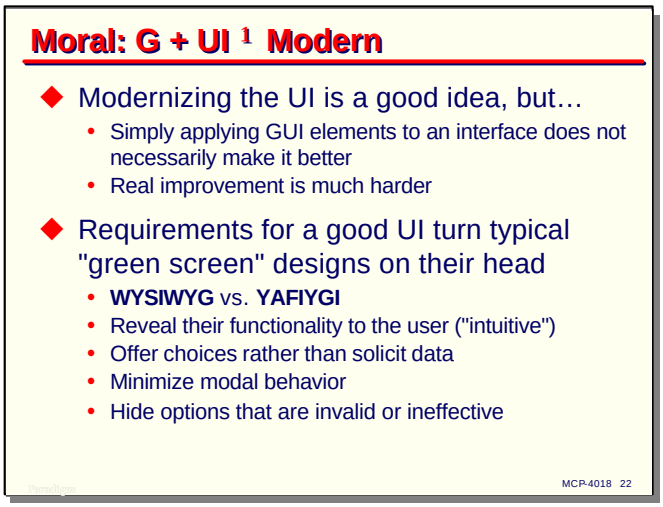
First, a façade is just that – a surface effect. To me, it misses the main point of modernization. As a result, it also misses most of the potential benefit.

Because it's just a surface effect, appreciation of it will probably be short lived. UI changes like this are largely cosmetic, and cosmetics are subject to fads. Any feel-good effect following implementation will wear off, and probably fairly soon.

Another thing to consider is that, having papered over the existing UI, we have done nothing to make the application more agile, or to prepare it to support additional interfaces in the future.

Something to be very careful about when implementing a UI façade is what I like to call "bifurcated implementation." The application is now implemented in two places. Depending on the distribution of talents among your staff, different groups may be working on the different pieces. This means that changes to the application must be applied in more than one place, and implementation of those changes must be coordinated. This can significantly complicate maintenance. On top of that, GUI elements such as choice lists will need to be kept up to date, along with data validation rules.

As I mentioned before, be very careful of moving business rules into the UI front end – that can lead to a maintenance nightmare and serious synchronization problems between the parts of the application.



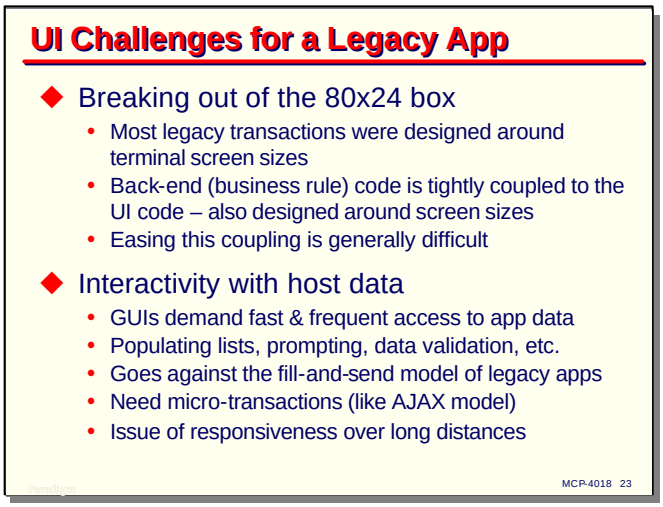
The moral of this story is that just adding a "G" (for graphical) to your UI does not get you a modern application. GUI elements are very nice, but by themselves they do not necessarily make an application better. A good UI is much more than graphical. What is more, trying to deal with the requirements for a good UI turns typical green-screen designs on their head.

Probably everyone is familiar with the term **WYSIWYG** – "what you see is what you get" – and associates it with modern GUIs as represented by Apple and Microsoft products, among others. What we need to appreciate is that WYSIWYG relies on a lot more than just the GUI – it requires an underlying design and organization of the application itself.

Most legacy applications simply are not designed for WYSIWYG. Instead, they are designed the old way, for **YAFIYGI** – the "you asked for it, you got it" approach. Here are some of the differences:

- WYSIWYG applications reveal their functionality to the user. This is most often done through a combination of pull-down menus, really good design of the dialogs with the user, careful captioning, tool tip pop-ups, and the like. This is where their "intuitive" behavior comes from. YAFIYGI applications are imperative – you have to tell it what you want, and you have to know at all times what it's listening for.
- WYSIWYG attempts to offer the user choices; YAFIYGI makes you enter the data it needs.
- WYSIWYG attempts to minimize user "modes;" YAFIYGI is often extremely modal.
- WYSIWYG hides controls that are invalid or ineffective so you can't use them by mistake; YAFIYGI requires you to know what's valid and what's not at all times.

Thus, it is usually very difficult to simply cover over a YAFIYGI-based design with a good WYSIWYG user interface without going through a lot of contortions. These contortions may include having to duplicate large parts of the application business rule logic in the UI front end, which we already know is not a good idea.



Beyond the disconnect between WYSIWYG requirements and YAFIYGI designs, there are a couple of real challenges to implementing a better UI for a legacy application.

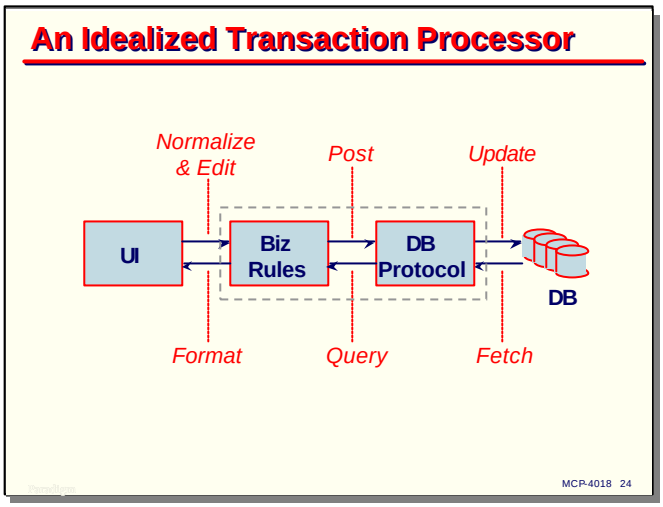
The first of these is what I think of as breaking out of the 80x24 box. Most legacy user interface devices (i.e., terminals) have a fixed window – 80 characters wide by 24 lines deep. This fixed size has had an enormous effect on application design, not only in terms of screen layouts, but also in the internal organization of the TPs that were responsible for those screens. I remember one of the big issues in the '70s and early '80s was how to handle data entry when there was too much source data for a transaction to fit on one screen. GEMCOS, the precursor to COMS on the A Series, had a mechanism for multi-page forms just to handle this problem.

Facilities such as the GEMCOS multi-page forms aside, most transaction screens were designed within this limited box, regardless of the nature of the original source. The screen handling code in the application TPs was of course designed to match what was on the screen, but since the back-end, or business rule, logic is typically intermixed with screen handling code, this means that the business rules and data base protocol implementations also got chopped up based on the 80x24 box. Once the UI and business rule coding gets tightly coupled like this, easing that coupling can be, and typically is, quite difficult.

Therefore, if you are trying to improve the UI for the application, you are sort of stuck with an underlying implementation that only knows how to feed you data in one way. This can be an issue for update transactions, but it is more often one for queries, especially ones that can return long lists of results. Searching for a customer named "Smith," for example, can yield a lot of candidates. The data base operations to generate such a list are very straightforward, but a lot of code had to be written in the TPs to break up the list into groups of 20 or so that would fit on one screen, display that group, and support continuing with the next group if the user wanted to do so. GUIs allow you to display lists of arbitrary length quite easily, and the UI takes over the job of scrolling items in and out of the user's window. We would probably like to take advantage of that with a new UI for our legacy application, but the underlying code only knows how to deliver results that fit in the original 80x24 box. There are several ways to deal with this, but the UI is now in a position of not just presenting the application's data, but *overcoming* how the application is trying to interact with the user.

The second major challenge for a new UI is that most GUI designs demand fast and frequent access to application data – they are much more interactive with the rest of the application. This is needed to fill choice lists, prompting the user with defaults or recently-used values, and for incremental data validation. It goes completely against the fill-and-send model used with screen forms, where the user is essentially off-line except when they press the transmit key. To support GUI designs, applications need to support micro-transactions, something like the AJAX technique that is currently revolutionizing HTML-based user interfaces.

Of course, doing micro-transactions and other typical GUI tricks within a local PC is no particular problem – the disk and app are in the same box. Doing large numbers of small transactions across a wide-area network may generate issues with responsiveness and processing overhead on the server.



I mentioned the back-end (business rule) logic and data base protocols on the previous slide. To illustrate the relationship among these and the user interface, here is my idealized view of a transaction processor. It is divided into three functional areas.

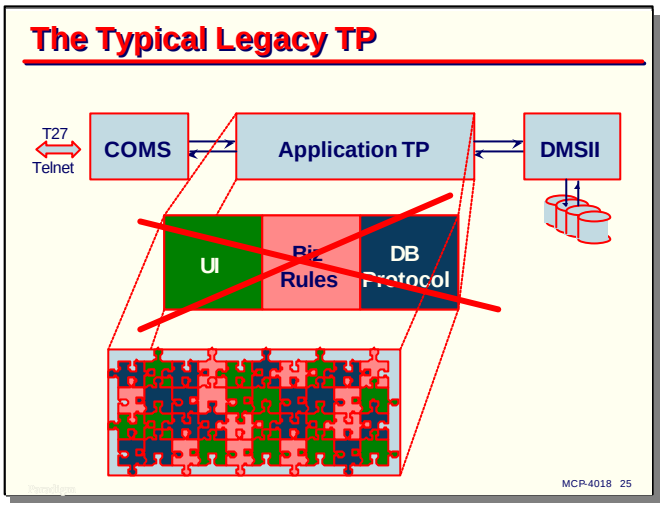
On the left is the user interface, which communicates with the end user. On input, it is responsible for normalizing the data (left/right adjustments, zero fill, decimal point alignment, standardizing case, etc.) and editing it. Generally, I think of editing as anything you can do that does not require knowledge of the application. This includes numeric editing, checking for valid dates, and possibly check-digit validation. Any validation that is specific to the application (e.g., account number ranges or code values) should be in the business rules and not in the UI. On output, the UI is responsible for formatting and representation of the application data to the user.

Next comes the business rules. I think of this as the "essence" of the application – what the application really does after you separate out all of the messy details concerned with interfacing with the user and interacting with the data base. Effectively, this *is* the application – all the rest of the code is just the plumbing necessary to make things work. The business rules are where most of the application value (and embedded knowledge) resides.

The third area is what I term the data base protocol. This is sometimes a little difficult to differentiate from the business rules, but I think of it as the process that maps the conceptual data items operated on by the business rules to the application's repository. While the business rules would be concerned with, say, a set of data items which comprise an account record, the data base protocol would be concerned with the physical representation of those data items, and their storage and retrieval in the data base. The DB protocol would also be concerned with such things as transaction commit and rollback. The business rules should not care what type of data base is being used, but the DB protocol would obviously be implemented differently for a DMSII data base compared to a Microsoft SQL Server data base.

There is necessarily a much stronger coupling between the business rules and the DB protocol than with the user interface. For this reason I usually think of the business rules and DB protocol as being co-located – either on the same system or on separate systems with a very wide, very low-latency pipe between them.

A very popular thing to do with relational data bases is to store significant portions of the business rules inside the data base, typically as stored procedures and triggers. This has some interesting advantages, but I am bothered by the way it splits the business rules and keeps them in separate places. This may have been necessary with thick-client designs where a large portion of the application resided remotely on the client, but I'm not sure it's appropriate for server-based applications.



With that view of an idealized transaction processor, let us look at a typical legacy COMS program. We have our communications monitor, COMS, on the left, which is responsible for interfacing to the network and routing data to and from a particular TP. We have the data base management software on the right.

If we look inside this legacy TP, conceptually we should see the three functional areas – the user interface, business rules, and data base protocol.

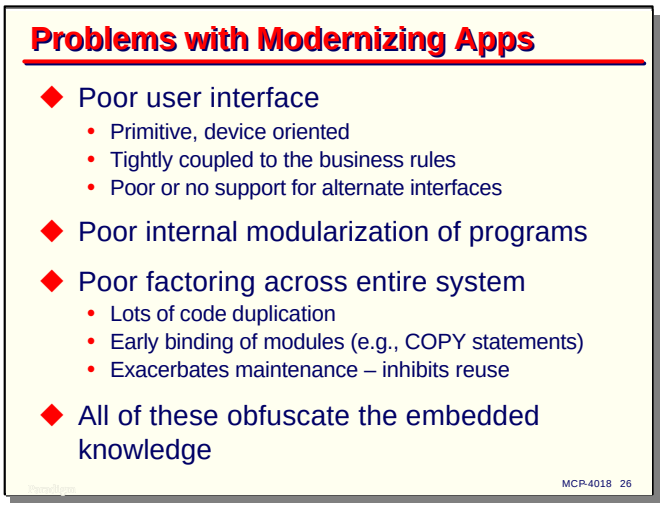
What we really see when we look inside most TPs, however, is not three clearly identifiable modules, but little snippets of UI code, business rule code, and data base protocol code all mixed together and very tightly coupled. This intermixing is often on a nearly line-by-line basis, so that without really studying the code, it's often difficult discern that the three functional areas exist.

It was never really necessary to design applications with the three types of coding all jumbled together, and I have seen many COBOL applications (including a few that I have written) which try to separate the three areas into two kinds of separate modules, but I have never seen one that successfully separated all three.

There are a number of reasons why most of our applications look like this. First, of course, was the problem of trying to implement on-line transaction processing using systems with very limited resources. Three megabytes of memory may have been a luxury for a batch environment, but it did not get you very far in supporting a large terminal network.

A second reason was that, until the advent of server libraries on the A Series, we did not have a good way to implement an application as separate modules and hook them together at run time. In some ways we still don't.

A third reason, and I think the major one, was simply expediency. Until fairly recently there has been very much of a get-the-job-done-don't-worry-about-tomorrow attitude towards application development. This attitude is certainly still with us, but some recently painful experiences, especially with Y2K and early deployment of thick client designs, are starting to teach us that we need to design and code for more than just the needs of today. 80x24 screens were all we knew, so 80x24 screens were all we designed for, just as with two-digit years. We now have to change that attitude.



To summarize this overview of the nature of legacy applications, here are some of the problems we can expect to encounter in modernizing them.

First, most have, by modern standards at least, a very poor user interface. This is due largely to the the limitations of primitive legacy terminal devices, but also because prior to the development of the PC, what we now call the user interface wasn't an interface for users at all – it was an interface for the computer.

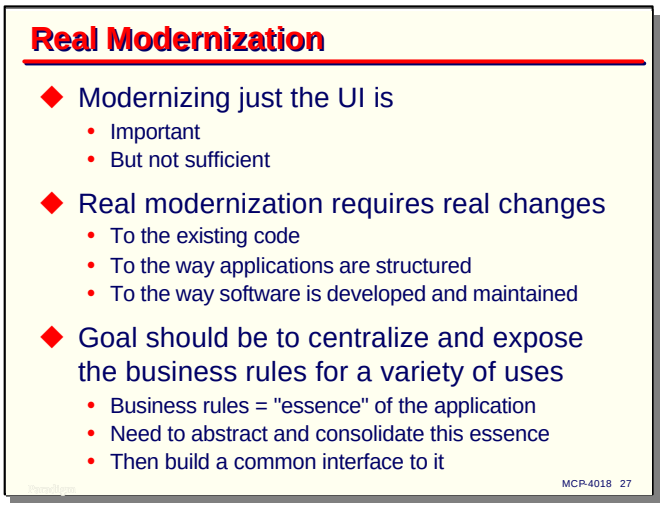
Second, in most applications the user interface is very tightly coupled to the business rules and data base protocol, so that it's very difficult to discern what is what. This makes it difficult to replace the user interface – there are simply a lot of interconnections that need to be sorted out.

Third, because the existing UI is so tightly coupled, there is usually either poor or no support for alternate interfaces to the business rules, say for web-based transactions or EDI. The usual solution here is to write an entirely new program that duplicates the business rules (or tries to) and creates another tightly-coupled jumble of code for the external interface, business rules, and data base protocol. Duplicating the implementation of business rules is a serious problem, because not only does it it increase maintenance effort, it is extremely difficult to keep the various implementations of the rules in sync.

Fourth, most programs suffer from poor internal modularization. This just makes the difficulties caused by tight coupling between the different parts of the implementation that much worse. A large part of this problem is the widespread use of COBOL. It requires real discipline on the part of a programmer to create and maintain reasonably modular and structured COBOL code. This is true to some degree with all languages, of course, but it is particularly true with COBOL, because it has such poor facilities for modularization and parameterization.

Fifth, looking the application as a whole, there is usually poor factoring of the code across the entire suite of programs. We normally see lots of duplication of code, from small issues like account number validation, to large ones like replication of the posting logic for whole transactions. I have worked on some banking systems, and have been amazed at the number of times the logic for, say, a checking account withdrawal is duplicated across the application, each one with minor variations to accommodate the type of source from which the transaction came or the conditions under which it was invoked. Where factoring has occurred, it is often implemented with early binding techniques, such as COPY statements or macro expansion. Changing the factored code requires identifying all of the places in the application where it has been used and rebinding each one.

Poor factoring and early binding exacerbate maintenance and enhancement of the software and contribute to reliability problems. They also inhibit code reusability. What they really do, however, is obfuscate the embedded knowledge and make it hard to access.

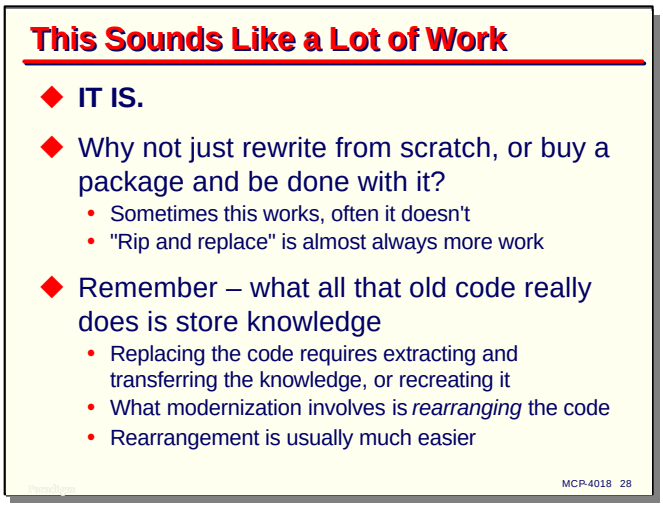


Modernizing the existing user interface to make it nicer for people to use is certainly important, but it is not sufficient. Modern applications have may kinds of users. Some are people, and those people have various types of needs and levels of skill in interacting with systems. Increasingly, however, some users are not people at all, but other applications. These applications could be on the same physical platform, or could be on a separate system, communicating over a network. The way you design an interface for people is a lot different than the way you do that when the other end is a piece of software. Those differences should not affect the implementation of the business rules and data base protocol.

This is the issue that I think is at the root of real modernization. Accommodating it requires significant changes to the structure of the application. That means changes to the code – major changes. It also means changing the way our applications are structured, and even the way we go about developing and maintaining those applications.

The primary goal of application modernization should be to centralize the business rules for an application and expose them in a way that they can be used for a variety of purposes. That is a process of abstraction and consolidation.

As the essence of the application, the business rules should not be concerned with the details of interacting with the external environment. That is the job of the user interfaces, which have the responsibility for translating the external representation of their environment to the common internal interface which the business rules should expose.



Making significant changes to the structure of an application to build an interface to the business rules seems like a lot of work.

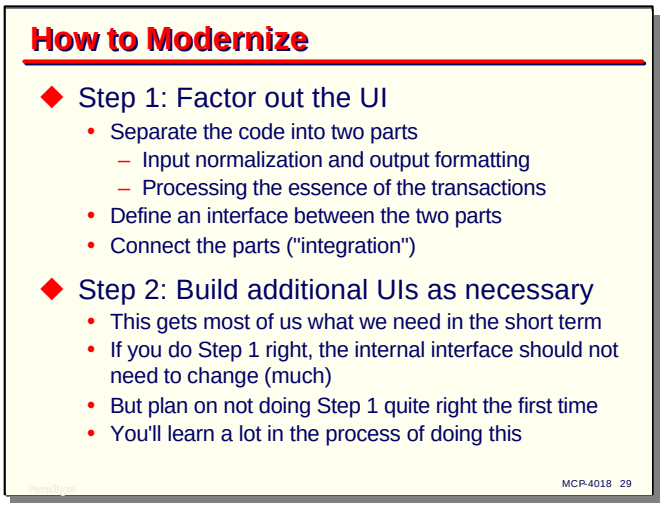
Yes, it is. For most applications it can be a huge job.

The temptation at this point is to conclude that it's too much work, and to simply dump the existing application and either rewrite it from scratch, or buy some package and install that. Sometimes this is the right thing to do – not every legacy application is worth preserving. In theory, at least, this approach can always be made to work. A lot of organizations have tried exactly this approach, especially as workstations and commodity servers appeared to approach the power of traditional mainframe systems. The track record for the rip and replace approach has not been all that good, though. Even when successful, it has almost always proven to be a lot more work than originally anticipated.

The thing that makes rip and replace look initially attractive is that we forget what all that old code in our legacy applications really does. It stores knowledge necessary to run our enterprises. In many cases, that old code is the only place that knowledge is stored, and often we are not really aware of its significance. We have to be careful that we don't throw out the knowledge baby with the COBOL bathwater.

Replacing a legacy application requires that we extract and transfer all that knowledge to the new application, or that we recreate the knowledge. As many have learned, this is a lot more difficult than it seems.

My view of modernization, however, primarily involves *rearranging* the existing code – factoring it, consolidating it, building interfaces to it – all activities that, while perhaps not trivial, are almost always easier and less risky to accomplish than wholesale rip and replace.



In my view, modernization needs to be done in phases, partly because there is just too much to be done to attempt it all at once, and partly because we simply do not know enough about how to do it yet. The way to get started, though, is quite clear. We need to deal with the fact that we now have a lot more types of external interfaces than the we used to.

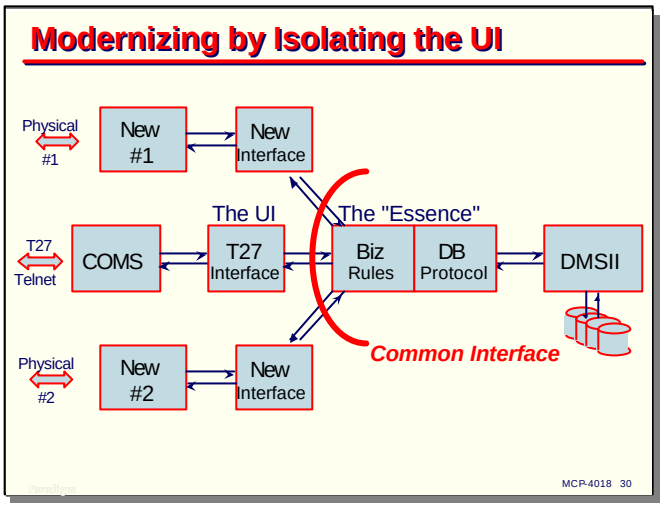
The first step is to factor out the user interface portion of the application and implement it separately. This requires identifying in the code which parts are concerned with input normalization, editing, and output formatting, and which parts are really dealing with the essence of the application. For brevity in the following, I'll lump business rules and data base protocol together and just call that the business rules.

Next, you need to define an interface between those two parts. Initially that interface will probably look like a data structure containing the data items from your existing screens, but once you get started on this, you should begin to see some opportunities for further generalization of that interface.

Once you have separated the code and defined that interface, you need to connect the two parts together. There are a number of ways to do this, some of which involve those nice integration technologies Unisys and others have been making available on ClearPath systems. I'll discuss this in the next couple of slides.

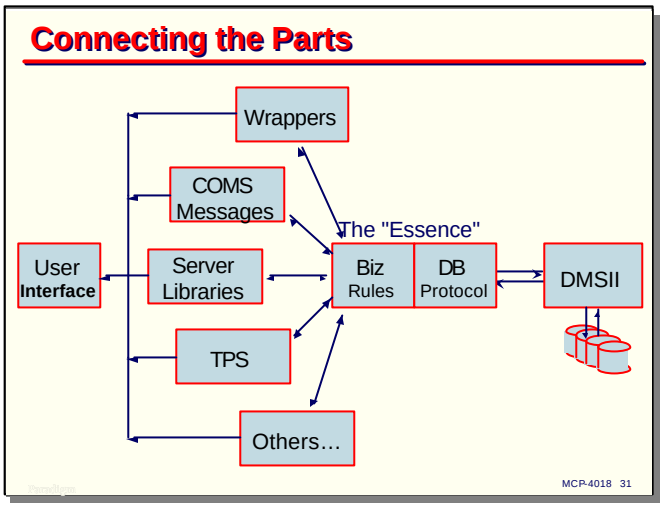
Now that you have a clean interface between your existing UI and the rest of the application, it should be a straightforward matter to implement additional external interfaces. Implementing additional external interfaces is what most of us need in the short term. If you did your job of designing the common interface in front of the business rules properly, that interface should not need to change as you implement additional external interfaces. I've often found, however, that new external interfaces require capabilities that were not needed by your original interface, and that these new requirements sometimes extend down into the business rules. In particular you may need to add the equivalent of micro-queries or micro-transactions to support a more modern UI implementation.

Doing this type of factoring and common interface definition takes some practice, so you should probably plan on not doing Step 1 completely right the first time. It would be best to start with a small system, or perhaps a well-separated subsystem of a larger system. Things like error reporting across the interface can be especially tricky, and your first attempt at the common interface may not be good enough. You'll learn a lot in the process of doing this, though, and the next time you will have a better idea of what needs to be done.



This diagram illustrates the basic idea of Steps 1 and 2. In Step 1 you factor out whatever legacy user interface you have (perhaps one for T27 terminals) and build a common interface between that and the business rules.

If you do a reasonable job of defining that common interface in Step 1, then the job of integrating new types of external interfaces to that common interface should be straightforward. Implementing an external interface itself may be a difficult task, but it need not be concerned with how the application works internally, as that is hidden behind the common interface. Most importantly, you will not need to duplicate business rule logic and later try to keep the multiple copies of it in sync with each other.



Having separated the external interfaces from the business rules and DB protocol, we need some way to connect them again so they can work together. ClearPath MCP systems provide a number of ways this can be done.

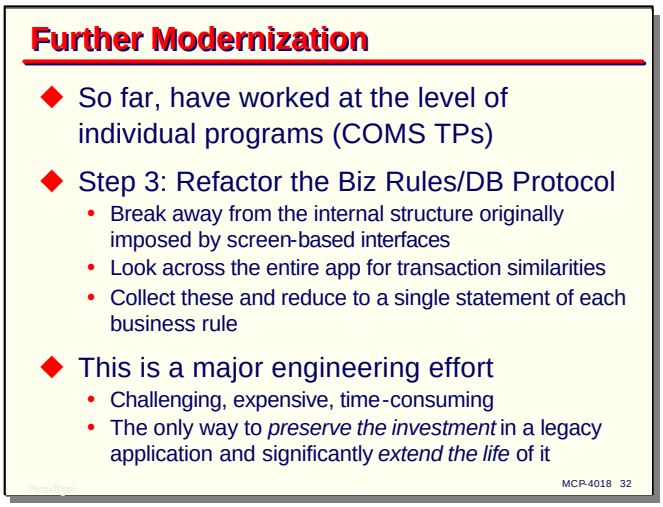
The server library facility of the MCP is an excellent mechanism for isolating and implementing internal transactions. You put the business rules and data base protocol in the library, and define transaction record layouts and library entry points to communicate with the internal transaction processes. You can then access these transactions from COMS modules, batch programs, XA/OLTP interfaces, XML data streams, and whatever else will be coming along in the future.

Unisys has a product, the Transaction Processing System, or TPS, that builds on program libraries to implement exactly this kind of internal transaction mechanism. It is a part of the DMSII suite, but is a separately-licensed product. You define transaction records in a DASDL-like language called TFL, and write libraries that implement business rules and data base protocols which operate on the transaction records. The TPS then generates libraries to connect and manage the flow of data between client tasks and the transaction libraries.

Aside from library-based methods there are a few other techniques that can be used to isolate the internal transactions from the user interface:

- You can separate the user interface and business rule code into separate COMS programs and communicate among them using standard COMS TP-to-TP messaging.
- You can also use the distributed transaction processing (XA/OLTP) facility in the MCP to isolate transaction code and provide multiple user interfaces to it.
- You can use port files to separate user interface code from internal transaction code in much the same way you can do it with COMS TP-to-TP messages. Another option is the use of the Core-to-Core and Storage Queue mechanisms inherited from V Series systems.
- If you choose to implement the user interface on an external system, there are now a number of "wrapper" or "connector" technologies that can be used to move the transaction streams into and out of the MCP environment. This includes COMTI, MQ messaging, and the Java connector.
- There are methods other than the ones mentioned here, and I expect development in this area will continue over the next several years.

Note that in this scheme the integration tools go *between* the user interface modules and the business rules, not on the outside of the application's existing (and typically, embedded) user interface.



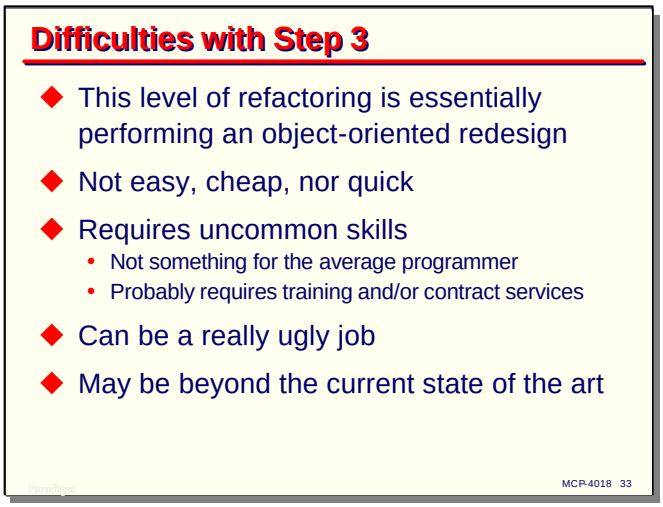
Steps 1 and 2 operate primarily on the original transaction processing programs. Modernization should not stop there.

The next step is to refactor the business rules and data base protocol code across the entire application. The idea here is two-fold:

- Break away from the internal structure that was originally imposed by the legacy screen interfaces
- Look across the application for functional similarities in transactions and condense those into a single implementation. In other words, try to state a given business rule just once.

This is largely a process of eliminating duplication of code (or enabling its reuse, whichever way you like to look at it). The most difficult part is recognizing the duplication, resolving the differences that will probably occur in each area where the rule is currently implemented, and defining a common interface to it.

Overall, you should expect this phase of the process to be really challenging, expensive, and time-consuming. I am convinced, however, that it is the only way to preserve the investment in a legacy application and significantly extend the life of it.



Talking about refactoring and consolidating business rule logic across an entire large application is a brave thing to do, and obviously this is a huge undertaking.

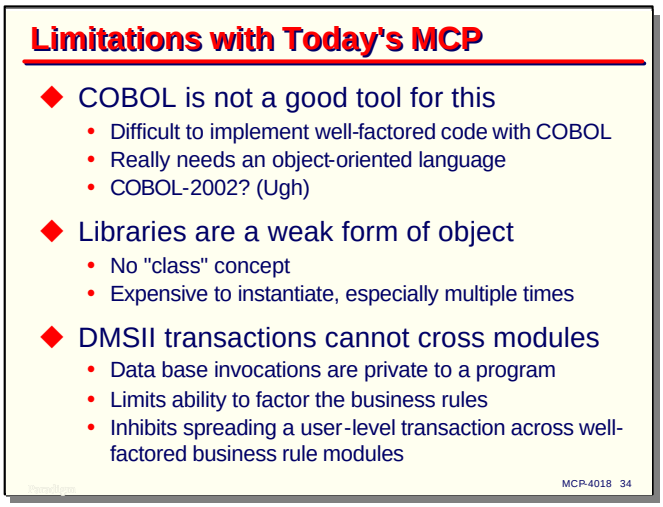
A Step 3-style refactoring is essentially the same as performing an object-oriented redesign of the application.[1] This is actually a good thing. The refactoring effort is all about abstraction and modularization. O-O technologies are the best way we currently have to implement that. The not-so-good thing is that the refactoring definitely will not be easy, cheap, nor quick to accomplish.

A major problem with this type of activity is that it requires uncommon skills. It involves the functional analysis and decomposition of large bodies of code, and then the synthesis of that into cohesive modules with attention to minimizing the coupling with other modules. Doing this for a single COBOL program is one thing – doing it across an entire application that has hundreds or thousands of programs is quite another. It is probably beyond the skill level of most journeyman programmers. It will probably require both training and contracting with outside services for assistance.

For most applications, I suspect this type of global refactoring will be a really ugly job. It is much, much larger and more complex than Y2K remediation was, and lest we forget, that was plenty difficult. Most legacy applications were just not built with this kind of internal organization in mind, and are likely to prove to be difficult to analyze and restructure in the way I suggest.

Finally, I'm not at all confident that we know how to do this yet with any assurance of success. It may be something that is currently beyond the state of the art.

[1] David Grubb, private communication.

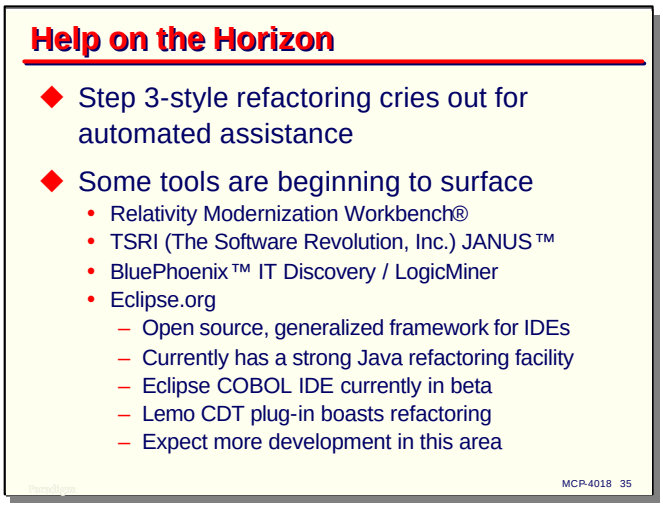


The inherent difficulties of a cross-application refactoring aside, there are a few significant problems with attempting to implement a set of well-factored applications on current MCP systems.

First, it is difficult to implement well-factored code in COBOL. COBOL-85 certainly has better support for this than COBOL-74, but COBOL is inherently a language for implementing programs, not systems of programs. Implementing a well-factored application cries out for an object-oriented language. Perhaps the new COBOL-2002 standard will be of some help here, but doing a good job of refactoring an app and converting it to COBOL-2002 appears to be about as much trouble as refactoring it and converting it to, say, Java. Besides, I suspect that COBOL-2002 will succeed in doing for object-oriented programming what COBOL-85 did for the subroutine call – make it so cumbersome that no one will bother to use it properly.

Second, libraries are perhaps the best modularization mechanism we currently have for MCP systems, but libraries are a very weak form of object. They are primarily just collections of subroutines, with some limited ability to maintain a data environment. They do not support any form of "class" concept. Since they require at least one hardware stack for each instantiation, they are very expensive in terms of system resources. They are not well-suited to environments which require multiple instantiations. We need a mechanism that can support tens of thousands or hundreds of thousands of "objects" easily and efficiently. Connection libraries go partway towards resolving these problems, but connection libraries require Algol or NEWP as languages, and still have problems with class and instantiation.

Third, a single DMSII transaction cannot presently cross any type of module boundary (other than lexically-bound modules, of course). Data base invocations are private to a program (or library), and transaction state is private to an invocation. Data base transactions need to correspond to user-level transactions, but many user-level transactions consist of collections of smaller, business rule-level transactions. If these are properly factored, some user-level transactions are going to need to activate business rules that are implemented in separate object modules, and there is currently no way to do that with DMSII. This also raises the potential for needing nested transactions, so that the individual business rule-level modules can back out of an unsuccessful operation without affecting the success of the overall transaction.

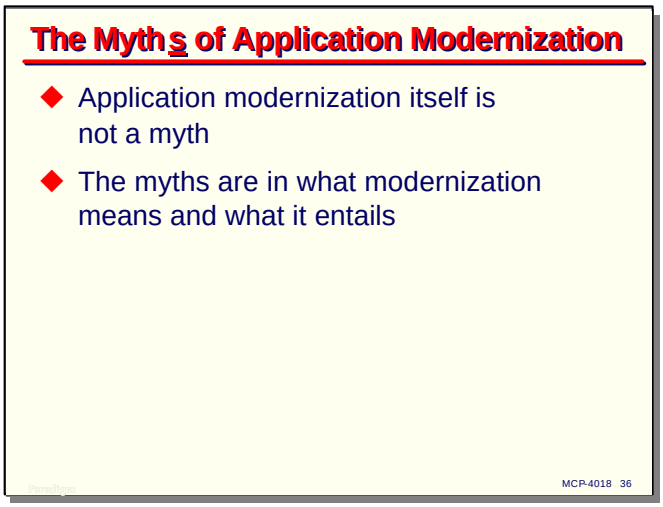


The type of analysis required for a Step 3-style refactoring cries out for automated assistance, and lots of it. As necessity is the mother of invention, some tools are beginning to become available in this area. This is just a sample list – I do not have any experience with any of these and cannot speak to their real capabilities.

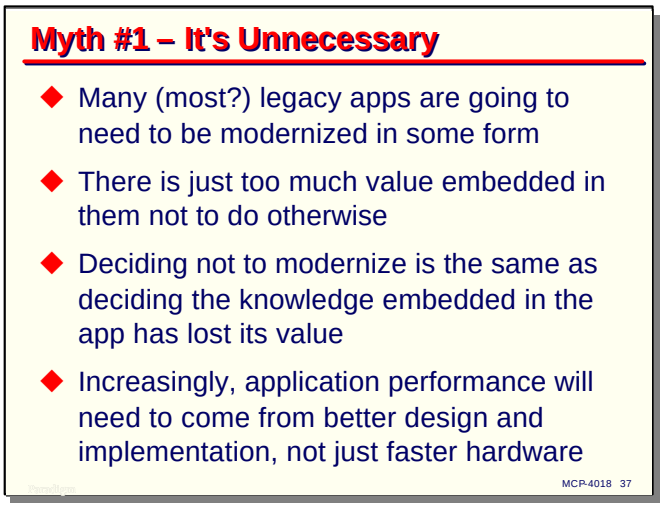
The most interesting of these to me is Eclipse. This is an open-source framework for IDEs, largely funded and developed to date by IBM. Its initial implementation has been oriented to Java, and the current implementation has a strong refactoring facility for Java applications.

The really interesting thing to me is that there is an Eclipse project devoted to developing a COBOL IDE. This has just recently entered beta testing. It does not currently advertise a refactoring facility, but there is an Eclipse plugin from a company named Lemo Soft (http://www.lemosoft.com) that does mention refactoring.

As recognition of the importance of preserving the investment in legacy applications rises, I would expect a lot more development in this area, especially from the open source movement. Stay tuned.

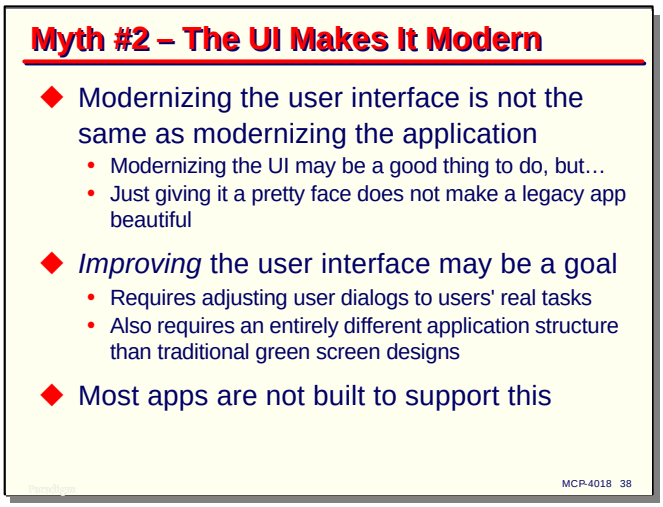


So, where does all of this leave us? Despite the somewhat provocative title of this presentation, application modernization, of itself, is not a myth. The myths are in what modernization means and what it entails.



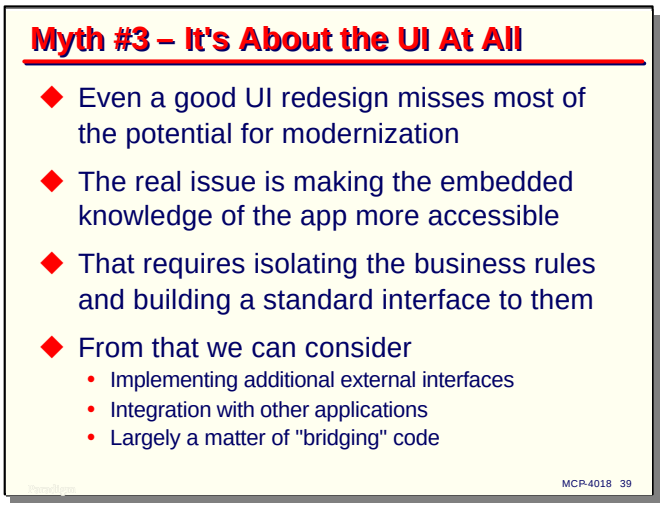
The first myth I'd like to dispel is that modernization is unnecessary. There may be a few applications out there that can continue living in the '70s and '80s, but most of us need to come into the new millennium and open up our systems to interaction with new user interfaces and other application systems. There is just too much value embedded in these applications not to attempt to preserve that value by some form of modernization. In fact, deciding not to modernize is essentially the same as deciding that the knowledge embedded in the application has lost its value. If that is the case, the application should be abandoned or replaced.

If I am correct about the coming demise of Moore's Law, then the free ride we have been getting on improvements to application performance is coming to an end. This is not an effect that is 20 years out – I think we will start seeing it inside of five years. Hardware performance will continue to increase, but not at the rate we have been accustomed to, and at a rate that will diminish over time. This means that at least some portion of additional performance improvement will need to start coming from the software, and in most cases that means major changes to that software.



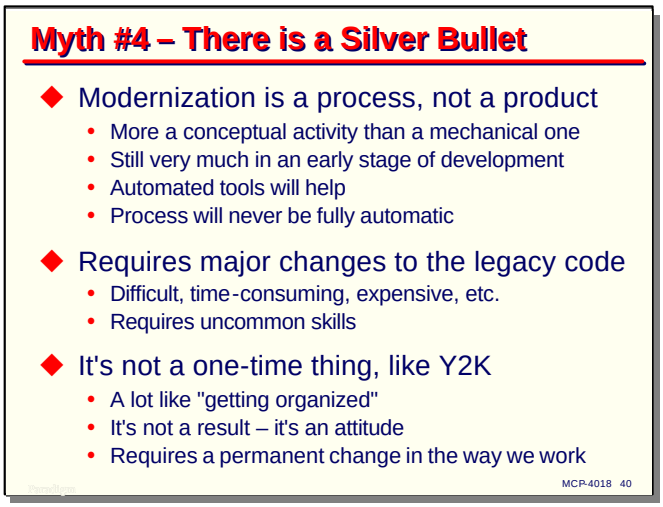
The second myth is that pasting a pretty face over your existing application is a form of modernization. If we are going to revise the user interface, we should be devoting our efforts to improving it, not just making it look nice, or "modern," or anything else superficial. That kind of improvement requires adjusting the UI to what the users really do. It also requires an entirely different structure and organization for the implementation than that used with most traditional green screen applications.

Unfortunately, most legacy applications are not built to support a really good UI. At least some refactoring, along with implementing additional support functions, will need to be done before even this type of improvement can be done.



The third myth is that application modernization is about the user interface at all. Certainly, improving the user interface and providing for multiple such interfaces may be one of the outcomes of modernization, but the real issue we need to face is making the knowledge that is embedded in our application more accessible.

Better accessibility requires that we isolate the business rules from the external interfaces and build a standard interface to those rules so they can be exposed to and accessed form a variety of sources. At that point, adding external interfaces or integrating with other applications is largely a matter of building "bridging" code between those external entities and our business rules – usually a relatively simple matter.



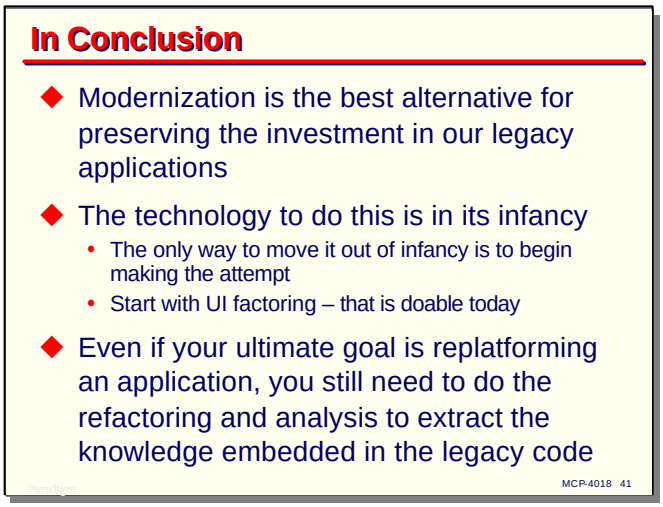
The final myth I'd like to address is that modernization is in any way going to be easy or pleasant to achieve. There is no silver bullet here.

Modernizing applications is a lot like modernizing your house. It requires tearing into things and rearranging them. You are going to be living with the software equivalent of plaster dust, sawhorses, plastic tarps, and raw floors for a long while.

Modernization is not a product. There is no complete, packaged modernization solution, and there probably never will be. Modernization is primarily about reorganization, standardization, and interface definition. These are conceptual activities, not mechanical ones, which means they will require talent and intelligence more than tools. The idea of transforming our applications in this way is a fairly new one, and both the concepts and the tools are in an early stage of development. Automated tools will certainly help, but the process will never be fully automatic.

Modernization requires major changes, in the form of refactoring, to the legacy code. That process, as we are currently able to do it, is difficult, time-consuming, messy, and very expensive. It will require skills that many of us do not currently possess. It will take time and experience to acquire those skills, whether we do it ourselves or retain others to help us with it.

Modernization is not a one-time effort. "Getting modernized" is a lot like someone saying they are going to "get organized." Getting organized is not a result, it's a change in attitude and behavior, one which requires an on-going commitment. Without that change, disorganization quickly reasserts itself. Similarly, getting "modernized" is not something we can just do, like Y2K remediation, and be done with it. Instead, real modernization involves a fundamental change in the way we conceive, analyze, design, implement, and maintain our application systems.



In conclusion, I offer that modernization is generally our best alternative for preserving the huge investment we have in our legacy applications. The technology to modernize large application systems is still very much in its infancy. No one is going to be able to solve this problem in the abstract. The best way to move the technology out of its infancy is by beginning to make the attempt.

Even if your ultimate goal is to eventually abandon your existing application and replatform – move it to a different system, or a different language, or whatever – you cannot escape the necessity of doing the refactoring and analysis I have discussed here as the basis for real modernization. That same work will be required to extract the knowledge embedded in your legacy code so it can be transferred, hopefully in a more useful and maintainable form, to the new platform.

For those of you interested in trying to work towards modernization, I recommend you start by trying to factor the user interfaces out of your current systems. That will give you an idea of what the larger process will be like. It will also be a good experience from which to learn more about recognizing the differences between the user interface, business rule, and data base protocol portions of existing code, refining the techniques used for refactoring, and defining interfaces for the refactored elements.

Good luck.

References

- Phillip G. Armour
  - "The Case for a New Business Model", *Comm. ACM*, August 2000, vol. 43, no. 8, page 19
  - "The Five Orders of Ignorance", *Comm. ACM*, October 2000, vol. 43, no. 10, page 17
- David Grubb, private communication
- Factoring tools
  - http://www.eclipse.org
  - http://www.relativity.com
  - http://www.softwarerevolution.com
  - http://www.bphx.com [BluePhoenix]
  - http://www.lemosoft.com [Lemo CDT for Eclipse]
- This presentation

  http://www.digm.com/UNITE/2005

End of
The Myth of
Application Modernization

2005 UNITE Conference

Session MCP-4018