

MODERNIZING LEGACY CA:2E & RPG APPLICATIONS

The concept of reusing existing code or logic is not a new one. The challenge has always been to identify, isolate, and reuse only those designs that are relevant in the new context in which they are desirable. In the case of IBM i, the sheer volume of code, its complexity, and the general lack of resources to understand legacy languages, specifically RPG and CA:2E, represent a tragic potential waste of valuable business assets for hundreds of thousands of companies. In many cases, these expensive and well-established legacy designs have little chance of even having their relevance assessed, let alone being reused.

To fully understand and appreciate the problem domain, just think for a minute of two approaches to the above problems namely: screen-scraping and code conversion.

Simply screen scraping the user interface with a GUI or web emulation product does not improve the situation, the application may appear slightly more ‘modern’ but the cosmetic changes still leave it with all the same maintenance and enhancement issues and it may be not much easier to use for new users. The same applies of building web services around wrapper programs written to interpret the interactive data stream from 5250 applications.

The other common approach is code conversion i.e. line by line, syntax conversion of a legacy application; this will typically just transfer the same problems from one environment/language to another. Indeed, it will often produce source code that is less maintainable, effectively canceling out the benefit of using modern technologies and architectures in the first place. Syntax conversions are still being done by some companies and are often promoted by vendors of proprietary development tools for obvious reasons. This approach has never to my knowledge produced an optimum long-term result, despite many attempts over the last two decades.

The objective, therefore in a true modernization project is to extract the essence or design of the legacy application and reuse these designs as appropriate in rebuilding the application, using modern languages, development tools, and techniques, and tapping into more widely available skills and resources.

In the previous three articles in this series, I described how to recover application legacy designs assets in a structured and proven manner. In this the concluding article, I will detail how to use these recovered designs to create a modern application.

MODERN APPLICATION ARCHITECTURE

Modern applications are implemented with distributed architecture. A popular standard used for this architecture is MVC or Model-View-Controller. Figure 1 below shows a typical legacy and MVC architectures side by side.

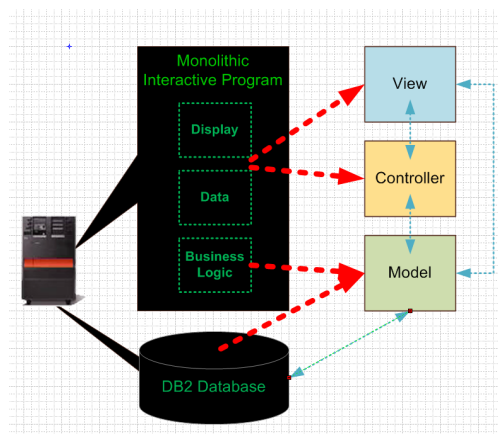


FIGURE 1

MVC allows for independent implementation and development of each layer, and facilitates OO techniques and code reusability rarely used in legacy applications. All these characteristics of a modern application radically improve the maintainability and agile nature. Legacy applications do have these same elements, but they tend to be embedded in and mixed up in large monolithic programs, with vast amounts of redundancy and duplication throughout. Implementing an RPG application using MVC requires that the business logic be separate from the user interface and controller logic. Figure 2 shows a schematic of the code implementation in a typical modern application.

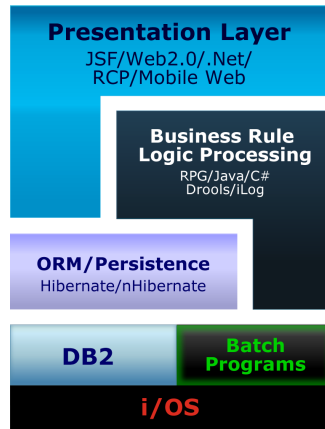


FIGURE 2

This architecture can be implemented using 5250 and pure RPG, but it's more likely and common implementation is using a web interface for the view, with the controller logic written in a modern language that supports web interfaces such as Java, EGL or C#. The optimum modernization result is to reduce dependency on legacy and proprietary languages as much as is possible, if not altogether if appropriate. To achieve this recovered design assets are reused as input to redevelop the appropriate layer.

Figure 3 shows an overview of the overall process of modernizing the legacy code using the recovered designs.

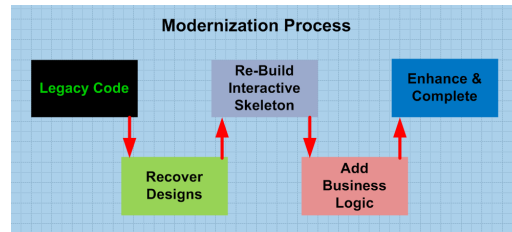


FIGURE 3

In a previous article “AUDITING LEGACY APPLICATIONS ASSETS” I discussed how to extract the data model and business rule logic from legacy code. In CA:2E applications this extraction is particularly accurate, relevant and clean if taken directly from the model. A CA:2E model is essentially an MVC specification of the application, and maps very neatly to modern MVC architecture. If these extracted designs can be articulated in language or programmatic format such as UML, DDL, XML, or even in a structured database, it is possible to use them programmatically to generate the basis of a new application skeleton. This can save companies millions of dollars and significantly reduce timelines. It also means that the designs can be perfected before any code is even written in the new application. Another benefit is that the generation process can be run repeatedly until the optimum start point of the new application development process is achieved, with very little effort and very rapidly.

This programmatic reuse of recovered application designs requires a certain amount of restructuring of the designs. The legacy designs of the interactive logic and flow of the legacy application can be used to build a modern application skeleton, and thereafter the extracted business rule logic added into this skeleton. Modern resources, tools and methods can then be used independently to enhance and complete the modernization as required. Let's look at these steps in more detail

BUILDING A MODERN APPLICATION SKELETON

The most fundamental change and biggest challenge in modernizing a legacy application, is moving from a procedural programming model to an event driven one. This aspect is one of the primary reasons that line-by-line syntax conversions to modern languages, produce results that are often less maintainable than the original code. One of the design elements in a legacy application that is almost directly transferrable to modern, event-driven programming model is Individual screen formats. Legacy screen formats largely if not explicitly correspond to individual steps in a transaction or business process. An individual web page or application form largely if not explicitly corresponds to an individual step in a transaction or business process. By simple deduction therefore, all of the design detail relating to the rendering of this specific legacy screen format can be used to specify and build a modern UI component. I will refer to the design information that forms this intersection as a “function definition” which is a standard construct in the CA:2E model as shown in Figure 4.

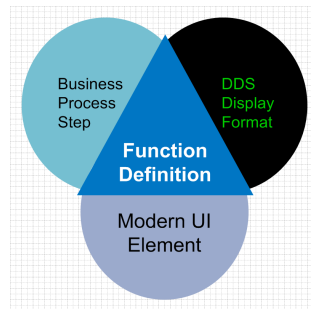


FIGURE 4

To rebuild a modern application skeleton from the legacy designs a function definition should consist of:

- Screen fields – work fields and fields directly traceable to database fields
- Screen field Types & Attributes – fields used as dates, foreign keys, descriptors, size type etc..
- Screen constants – column headings, field prompts, function name, command key descriptions etc..
- Screen layout – column and row positions can be converted later using relative pixel ratios
- Screen field database mapping – where the data for the screen comes from including join rules for foreign keys
- Screen Actions – command keys, default enter, and sub-file options

This design information is entangled in DDS, program logic and the database of the legacy application. With a reasonable level of skill in legacy languages, one can extract this manually by analyzing the source code manually. With larger systems it is advisable to use tools for the analysis and extraction process. The added benefit of using an analysis and extraction tool over and above productivity, consistency and accuracy, is that the results can more easily be stored programmatically and so be used to automate the next step of writing the code. UML is one way that this can be achieved. The function definitions can be generated as a UML model for an application with a number of specific UML constructs being used that will also assist in modeling and documenting the new application for modern developers. Some of these constructs include Activity Diagrams, Use Cases and Class Diagrams. Fig 5 shows a UML Activity Diagram that represents the users' flow through a series of legacy programs having multiple screen formats.

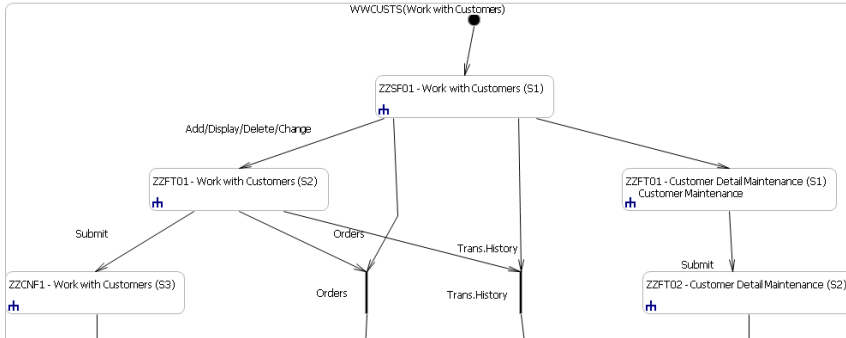


FIGURE 5

DDL and XML can be used as a means to efficiently specific the detailed aspects of the function definitions. DDL created from the legacy application data model can be imported into a persistence framework or Object Relational Map, such as Hibernate for Java and nHibernate for .Net. An ORM greatly simplifies the subsequent coding required in Java or C# by subcontracting all of the complicated SQL programming required in an enterprise business application. An additional approach is to create a single database I/O class for each table. This removes the need to have I/O logic embedded in every program in the system, this immediately making the application more maintainable and agile.

The function definitions are then be used to create the user interfaces and controller beans in the language and standard of choice (one JSF and corresponding Java bean per legacy screen format). Using XML to store the function definition provides input for both documented specifications for manual rebuilds, and as an input to programs that can create the view and controller components. This approach is applicable to Java, EGL, C#, and PHP implementations and can be used for web, mobile web, and Rich Client Platform alike. This is an important factor for enterprise applications that often require a mix of device types and even technology implementation options for a single system. Figure 6 below shows a JSF generated from a function definition that was extracted from a legacy program.

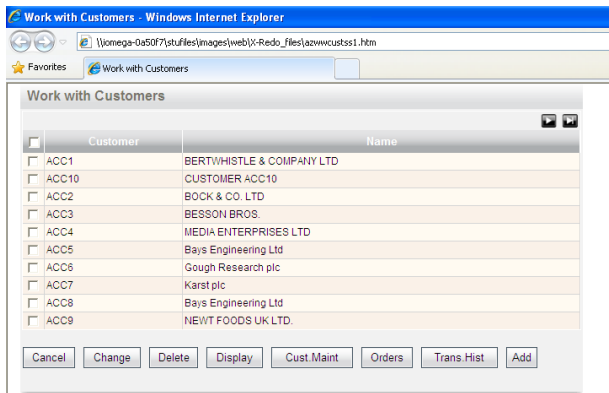


FIGURE 6

The important factor here is not so much the look and feel, but rather that each button is now associated with an event handler in the underlying JSF bean which is triggered by the HTML code itself. The data in the grid was retrieved from the DB2 for i database using the SQL in the bean created for the underlying database table, which was invoked by the JSF bean when the JSF page was requested by the user, in this instance from a menu on a previous page. The HTML and CSS layout was created by using the information from the function definition, as were what buttons to put into the JSF from the options, command keys and default enter, all extracted from the legacy program. In this instance the design was extracted from legacy RPG/DDS and the JSF and Java beans created both automatically using a tool in a few minutes. The style was implemented using a standard CSS file and supporting images. All industry standard, best practice modern stuff. Figure 7 shows a small snippet of the underlying HTML code which triggers an event in the bean to invoke the orders page, passing the key of the row selected by the user.

```

</h:commandButton> <h:commandButton id="ZZWWCONHDRS1" value="Orders"
title="ZZWWCONHDRS1" styleClass="buttons"
onclick="setGridTarget('_top')" disabled="{azwwcustss1.gridEmpty}"
action="{azwwcustss1.getAction}"
actionListener="{azwwcustss1.orders}">
<f:param name="XWBCCD" value="{azwwcustss1.XWBCCD}" />

```

FIGURE 7

The underlying Java bean knows what to do because the parameter being passed tells it where to go next. In this way the JSF beans can be kept small and simple; another good industry standard and best practice. Figure 8 shows the record type page that the user is taken to when selecting the record and the change button.

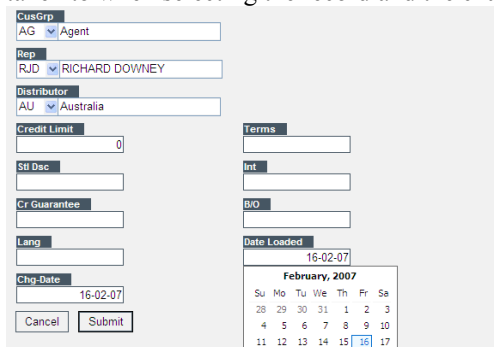


FIGURE 8

The drop down combo boxes and date controls were added because of the presence of the foreign key information, and date field types respectively in the extracted function definitions. This simple algorithm can save thousands of hours of configuration and editing of web pages in a modern application with hundreds or thousands of screen formats.

ADDING BUSINESS RULE LOGIC

In a previous article “AUDITING LEGACY APPLICATIONS ASSETS” I described how business rule logic could be extracted, indexed and documented from legacy RPG code. In CA:2E applications this business logic can be extracted directly from the action diagrams. One approach would be to add these documented rules manually to the appropriate business logic class in the modern application. This approach should be reserved for cases where very little of the legacy business logic is to be reused, including of course smaller programs that have little or no specific business logic beyond what has already been created in the JSF, JSF bean and

database I/O beans. It is important to reiterate here that the same principals described here using JSF and Java examples are exactly applicable in .Net and even modern RPG applications.

Another more practical approach which has already been automated is to essentially refactor the original interactive program, to the extent that only the business logic processing is reused. Naturally the refactoring must include restructuring to turn it from a procedural design to an event driven one. Again this process is applicable whether creating Java, .Net, EGL or even RPGLE business logic components. During the initial modernization effort, the business logic bean should be created as a single class/module/program that services each the modern event driven JSF's that came from the original legacy program. This is a maintainable architecture and follows modern coding practices, but retains at least some reference to the legacy transactions. Figure 9 shows a schematic representation of the architectural mapping between legacy and modern designs for a single legacy program.

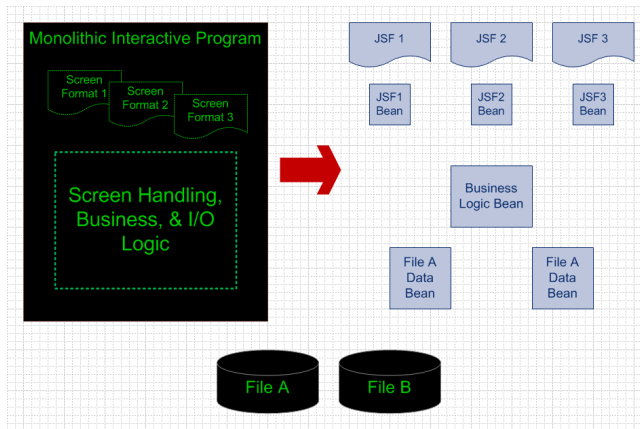


FIGURE 9

The original legacy program A had 3 screen formats, which become 3 discrete free standing JSF's and corresponding Java beans, 2 database I/O classes determined by the number of unique tables or physical files, and a single business logic bean/module/program for the business logic. I have used JSF and Java for this diagram but the same architecture is applicable for any modern language. The same architecture would be consistent with using Spring & Hibernate frameworks too.

The business logic bean now contains a restructured version all of the relevant business logic from the original program. This restricting process turns business logic from procedural into event driven, and in doing so maps the relevant business rules processing to the relevant JSF. The first step to achieving this restructuring is to recover the logic executed before screen 1 is rendered. This will essentially be placed in the pre-entry method or procedure of new business logic bean, and invoked by the JSF Bean before the JSF is displayed. Any legacy UI logic such as interactive indicators is removed during this extraction.

The next step is to map the business logic to each JSF. This is done by identifying the business logic that is executed after the legacy format 1 but before legacy format 2. Ignore the interactive logic, and legacy structures such as indicators, and turned into variables where applicable. This logic is then created as a new method (called JSF1 validation for example) in the new business logic bean. This business logic is invoked by the JSF bean that corresponds with legacy format 1, when triggered by the validation event in the new JSF. This is usually the submit button on the JSF itself. This stage is repeated for each of the legacy screen formats/JSF's. The documented, indexed business rules described in previous articles in the series, can be used as a reference for auditing the applicable logic during this refactoring and extraction exercise.

Finally the legacy subroutines by simple logic can be considered business logic and as such, have a scope that is potentially applicable to any of the newly created validation methods/procedures. Therefore only the legacy specific code or redundant interactive lines need be removed before these subroutines are code or copied into the new business logic bean. Figure 10 shows an example program outline documented in a form of platform independent pseudo code.

```

// Business logic for Customer Detail Maintenance (CUSTMNT1) //
PREENTRY processing
VALIDATION processing for screen format ZZFT01 (JSF1)
  Execute VALID1
  // If errors found
  If *INVALID
    // Display error message and re-display screen
  Else
    // Call program 'WCUSTP' passing the field SWBCCD.
    Call WCUSTP
      Parm Customer
  End
VALIDATION processing for screen format ZZFT02 (JSF2)
GETREC
UPDREC
VALID1
VALID2
WRTREC
ZGETNAMES

```

FIGURE 10

I have included only the method outline with one of them expanded and added some simple color coding. The pre-entry method will be executed before JSF1 is rendered, Validation for JSF1 when the user selects the submit button from the web page and so on. The GETREC through ZGETNAMES are subroutines that have had the interactive logic removed and verified to contain valid business logic in them. It is not possible in such a short article to show the complete detail, but detailed examples are available upon request.

The harvesting of valuable designs is now complete, and the application can now be enhanced, and refactored. It's worth noting that there are tools available to automate each of the steps described in this process. Staged automation of the recovery and rebuild process can reduce a system rewrite effort by at least 50%. Even executed manually this approach provides for iterative and parallel use of resources, and is applicable for individual programs, application areas consisting of multiple programs, and even entire systems. It allows for sustained reuse of legacy technology but is not bound by it, while simultaneously producing a real modern application, not an emulated one.

FURTHER READING

Here are some useful publications that cover almost all of the subjects discussed in this article:

- **IBM Redbook - Modernizing and Improving the Maintainability of RPG Applications**
<http://www.redbooks.ibm.com/redpieces/abstracts/redp4046.html>
- **Recycling RPG - Strategies to create new applications from legacy code**
<http://systeminetwork.com/article/recycling-rpg>
- **Recycle Your Legacy Code with Daborough's X-Analysis 8**
<http://systeminetwork.com/article/recycle-your-legacy-code-daboroughs-x-analysis-8>