Extending the Life Cycle of Legacy Applications

(With Added Thoughts Specific to Assembler Language)

SHARE 103 (Summer 2004), Session 8132

John R. Ehrman
ehrman@us.ibm.com or ehrman@vnet.ibm.com

IBM Silicon Valley (Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, CA 95141

August, 2004

Table of Contents

Table of Contents

Topic Overview

leg'-a-cy (*n.*) something handed down from an ancestor or predecessor or from the past

- "Legacy" code: Something of value; worth preserving; useful
  - What's good and bad about it? What's **thought** to be good and bad?
- Why do something with it? (And why ***not*** do something?)
- What can be done with it?
- Some suggestions and recommendations

- Some abbreviations:
  - **AL**: Assembler Language
  - **HLL**: High Level Language (e.g. COBOL, PL/I, C/C++)

Legacy code: language-independent issues

What's Bad	What's Good
Old	Stable; very low error rates; the continuing payback on past investments
Complex	Embodies the necessary complexities of the business and its environments
Incomprehensible	Business rules are often that way
Hard to learn, Hard to fix	The remaining bugs are rare, obscure, nontrivial
Dull, boring	Business applications rarely provide entertainment value (to their programmers, that is...)

Legacy code: Assembler Language issues

What's Bad	What's Good
Old	AL is by far the most stable host language; extensive modernizations are now available
Complex	New programming techniques and tools can greatly clarify the underlying (complex) business logic
Incomprehensible	Comprehension can be improved by • structure and adherence to conventions • adequate documentation and commentary • familiarity with machine architecture • familiarity with new assembler capabilities
Hard to learn, Hard to fix	Not necessarily inherent in AL itself; good documentation and new technology tools help a lot
Dull, boring	AL is a rich language (but not “technology du jour”)

- Legacy issues may (or may not) depend on a programming language

Assembler Language: what is *believed* to be bad?

- Structured programming is impossible (?)
  - Many structured-programming macro packages are available
  - Assembler Language is far more flexible, offers many more application structuring facilities than HLLs
- Maintenance is much more costly than with HLLs (?)
  - Research shows there's no language-related difference
    - Programmer ability was found to be the ***only*** significant factor
  - Obscure code can be written in any language
    - Some languages obscure their obscurities; AL can't
- Assembler Language is hard to learn (?)
  - True for some; but AL training has widespread value, good payoffs
    - Understanding machine architecture is very valuable
  - Modern tools/techniques enhance understandability, modifiability

Why change *any* language to another?

Reasons ***for*** changing computer language include

1. Skills availability
2. Standardization on fewer (or one) language(s)
3. Maintainability
4. Application portability to a wider range of hardware platforms
5. Belief that a new language is “Industrial Strength”
    - Improved productivity, maintainability, widespread use, etc.
6. Bulk buying discounts
7. Storage-constraint relief
8. Fashion, competition (“Our competitors seem successful using language X and system configuration Y”)

We'll review these concerns, later...

Changing languages: why not?

Reasons for ***not*** changing computer language include

1. Cost, time, and complexity of conversion effort
2. Quality and maintainability of converted code
3. Continuity of planning, procedures, policies, and expertise
4. Hardware upgrade costs
5. Cost of new software products (compilers, run-time libraries)
6. Loss of flexibility
7. Performance concerns in the new language
8. Loss of function
9. Re-training costs
10. Language longevity

We'll return later to these concerns, also...

What can be done with an assembler legacy application?

- We will discuss five typical scenarios:
  1. Live with it (“Business As Usual”)
  2. Replace it with a vendor's commercial package
  3. Use automated conversion tools to generate “equivalent” HLL code
  4. Rewrite it in your favorite HLL
  5. Modernize it and continue to use it
- ... considering
  - Organizational and managerial concerns
  - Staffing and skills
  - Technical issues
  - Financial factors
- These topics cover the main issues

(1) Living with existing legacy application code

- Living with the current code may be the best choice, if:
  - Requirements for enhancements don't have widespread impact
  - The code is adequately structured, documented, and commented
  - Available support tools are sufficient for your needs
  - Current staff can handle the requirements
  - Risks (time, money, disruptions) of other alternatives are greater
  - Continuity is important
- Some organizations have lived this way for a long time
  - Allows you to avoid the latest programming fads
  - But some may want to do things differently (“better?”)

We'll reconsider this Scenario in Scenario (5)

(1) Living with existing legacy application code ...

- Organizational/managerial concerns: Business as Usual
  - Minimal internal and external impact on business
  - Smaller incremental costs
- Staffing and skills: Business as Usual
  - Existing code/test/planning/maintenance procedures needn't be changed
    - No retraining needed
  - No decrease in productivity
    - New tools can improve it
  - Skills pool may evaporate slowly
    - Internal training, mentoring, and education may refill it adequately

(1) Living with existing legacy application code ...

- Technical issues: Business as Usual
  - Virtual storage constraint relief easy to do without major changes
    - Buffers and work areas above 16MB; split-RMode modules
  - Converting to Language Environment usually straightforward
  - Staging of updates easy to do incrementally
    - No need to stabilize the current application to do a substitute
    - Minor fixes easy to apply quickly
  - Substantial enhancements may be difficult to accommodate
- Financial factors: Business as Usual
  - Slow enhancements may lose market opportunities
    - May motivate investigating other scenarios?

(1) Living with existing legacy application code: summary

- Simplest, cheapest short-term solution
- Minimal impact on planning, procedures, staff, end users, and morale
- Minimal requirements for new staff, training, and tool sets
- Stable code, known test and maintenance procedures
- No costs of new hardware or software, or training
- No damage to corporate reputation
- Concerns: lost business opportunities, staff retirement

(2) Replace the old code with a vendor package

- Someone else proposes to do the “dirty work”
  - “No more problems with our unresponsive IT staff...” (?)
- General-purpose packages normally require customization
  - One size doesn't usually “fit all”
  - Your business vs. the package: which adapts to the other?
- Organizational/managerial concerns:
  - Detailed acquisition specs may be needed
  - Extensive planning typically required
  - Ensuring package source-code escrow
- Staffing and skills:
  - Added burdens on existing staff
  - Train new support staff, establish vendor interfaces
  - New support procedures, internal and end-user training

(2) Replace the old code with a vendor package ...

- Technical issues:
  - Transition to new system a major effort
  - Extended testing and parallel use of old and new systems
    - Old data may need frequent reformatting and transfer to new system
    - Cut-over to new system only after all functions are certified
    - Expect to retain old system and data for archive/legal requirements
    - New package may not perform as well as the old application
- Financial factors:
  - Vendor charges for design changes ($$)
  - May require new/upgraded hardware; software charges may increase
  - Probable consulting and tailoring expenses ($$)
  - Training materials may be proprietary ($)
  - Cost of maintenance contracts ($$)
  - Possible lost business opportunity during transition period
    - And, during production-use problem-reporting and problem-repair periods
  - Estimating overall costs likely to be very difficult

(2) Replace the old code with a vendor package: summary

- Difficulty of assessing vendor skills
- Long transition period
  - Documenting existing and desired functions
  - Extensive parallel testing
  - Possible requirement for data conversion
  - Possible lost business opportunity
- Integration/customization costs sometimes 5-10 times package cost
- Vendor charges for training, code modifications, maintenance
- Possible hardware upgrades
- Possible staff morale problems
  - Retraining requirements; loss of responsibility, skills, competences
- Possible lack of market differentiation

(3) Convert languages using automated tools

- This option is **not** recommended, but is worth reviewing...
  - Hoped-for benefits are rarely achieved
- Some vendors advertise conversion tools and services
  - Conversion technology is still very immature
  - Academic efforts don't address the messy “real world” very well
  - Poorly structured/commented programs don't convert to useful code
  - Significant manual intervention may be required
- Many factors to consider
  - Syntax conversion is only a first, small step to convertibility
    - Necessary code-preparation effort may be substantial
  - Resulting programs generally unreadable, inefficient
    - Require deep knowledge of the original **and** the target languages (and maybe the source and target hardware architectures)

(3) Requirements for conversion tools and procedures

- Inventory language constructs, develop conversion strategy for each
  - Specify the allowable level of pre-conversion “manual clean-up”
- State the required degree of functional equivalence
  - ...and whether code must more resemble the old or the new language
    - May depend on language experience of maintenance/development teams
  - Specify whether or not test sets must be converted
- Decide how to handle cases that aren't converted automatically
  - ...and who is responsible for fixing them
- Determine result performance, size, and maintainability requirements
- Specify allowable resource utilization by the converter
- Don't expect too much, or too soon, or ...

(3) Possible problems with automated conversion

- Poorly structured old code will become worse-structured new code
- Converted code's language may resemble neither source nor target
- Statement-by-statement converter results aren't very helpful
- Results from converted code may not agree with original program's
  - Plan for exhaustive testing; existing test suites rarely suffice
- Size and performance of converted code may be unsatisfactory
- Some AL idioms and instructions may not convert easily
  - May need (manual) repair work
- Distinction between “true” constants and initial data may not be apparent

(3) Automated conversion considerations

- Organizational and managerial
  - Significant effort to establish accurate conversion requirements
  - Phasing of test, acceptance, cut-over to new applications
  - New procedures for planning, estimating, scheduling, development and test, product delivery, maintenance
  - Political procedures for dealing with unsatisfactory results
- Staffing and skills
  - Need people skilled in old and new languages
  - Training required in new procedures
  - Significant regression-testing effort
  - Old-code preparation time/effort might be resented

(3) Automated conversion considerations ...

- Technical issues
  - Pre-conversion code documentation, mods (and fixes!)
  - Incomplete conversions requiring manual intervention
  - Resulting code looks like neither old nor new language
  - Increased size, decreased performance of results
  - Some AL function not expressible in target language
- Financial factors
  - Staff resources reassigned to conversion activities
  - Converter (and vendor) expenses
  - Hardware impacts of slower, fatter code
    - May need additional hardware for testing while production continues
  - Costs of new HLL and development environment software
  - Lost-opportunity costs

(3) Convert to a HLL using automated tools: summary

- Conversion tools help only “mechanical” aspects of translation
  - A “finished product” requires extensive hand work, deep knowledge of original and target languages
  - OS and language differences may be significant
    - Data types, data structures, file systems, system services, error handling, ...
    - Some language idioms are hard to translate to target language
- Created code may grow larger, run more slowly
- Lengthy parallel execution and testing
  - Detecting and correcting inaccuracies can be ***very*** difficult
  - Convert existing test suites? Write new test cases?
- High levels of organizational stress
- Additional conversion concerns discussed in Scenario (4)

(4) Rewrite the application in a new language

- This is a widely considered scenario, with anticipated advantages:
  - Standardize language, utilize skills, ... (as noted on slide 5)
- Converting entire applications is difficult
  - Much more difficult when moving to a new platform
  - Enabling LE compatibility of small AL routines is straightforward
- Requires very careful investigation, planning, preparation
  - We'll look at some details worth investigating
- Results may be disappointing
  - “Problems to be solved by conversion are usually replaced by other (perhaps more intricate) problems”
- Two references are highly recommended (see slide 44)

(4) Choice of target language

- This may not always be obvious!
  - Choice of language has little effect on productivity, maintenance costs
  - Recommended languages change with IT-industry fashions
  - Specialized languages are less flexible; general languages may be complex
- Language must support correct data types, computational behavior
- Similarity of old/new syntax/semantics may not help
  - Important differences are much harder to detect
- Dissimilarity of old/new syntax/semantics a possible problem
  - New language may be inflexible
  - Excess of features easy to misuse
- PC-popular languages may not be appropriate for mainframes

(4) Preparing for rewriting in a new language

- Document ***what*** the application does: you'll need it first
  - What business rules are implemented by the code?
- Document ***how*** the application does what it does
  - Re-structure and (re-)comment the source code
    - Existing documentation may be unreliable
    - Make sure names are meaningful!
  - Determine internal/external data, user-interface interactions
  - Note potential convertibility problems
  - Isolate system-service interfaces to a single module
- Assess requirements for changes to support tools, procedures, etc.
  - Organize and verify test suites for before/after testing
  - Plan for extensive restructuring **before and after** conversion
- Document what the application does, **and** why: you'll need it later

(4) Program analysis, understanding and restructuring

- Several tools can help with AL (see slide 35)
- For single or linked modules:
  - A graphic “program understander” to display control flows
  - A symbolic debugger to track code and data flows
- For inter-module control flows, shared-data references
  - A source, macro, INCLUDE/COPY-file cross-referencer
- Look for “macro-instruction” opportunities
  - Encapsulate recognizable HLL-like actions, rewrite them as macros
    - looping, if-then-else, conversions, repeated clichés, ...
- Be sure to annotate and document the “Why” factors (and the “Why Not” factors!)

(4) Convertibility and conversion problems

- Target language characteristics
  - Do programs require a “main” entry?
  - Do local variables retain last-used state?
  - Are overflow and other exception conditions detected? Are they correctable?
- Data representations and structure mappings
- Internal and external control flows
  - Linkage, parameter passing, status preservation conventions
- Assembler Language usage
  - Specialized instructions
  - Conditional assembly and macros
- Considerable rewriting may be required

(4) Convertibility and conversion problems: data

- Verify correct mappings of all
  - data types: lengths, representations, character strings
  - data structures: arrays, structures (alignment, gaps)
  - immediate operands
- Check for operations involving internal data representations
  - character encoding, collating sequence, and byte-order dependences
  - type equivalences
  - sign representations and sign positions
  - address arithmetic
  - logic testing of multiple bits per byte
- Verify recognition and handling of data-exception conditions
- Watch out for (probable) side-effects

(4) Convertibility and conversion problems: control flow

- Procedural logic may utilize knowledge of data types/layouts
- Control flow may not easily map to HLL statements
  - HLL may not express or support Assembler Language program structures
  - Assembler Language programs use very flexible structuring and call/return mechanisms
- Status-preservation rules may differ
- Argument-passing and value-return mechanisms
  - Procedure status indicators in registers or storage
- Ensure correct mappings for data shared among modules
- Check for exception signaling and handling differences

(4) Convertibility and conversion problems: Assembler Language

- Hardware architecture differences
- Multiple branches following a Condition Code setting
  - Especially if the CC-setting instruction has side-effects (TRT, EDMK, ...)
- Code with absolute base/displacement assignments
  - May require changing instructions, data declarations
- Self-modifying code
  - Yes, it still happens from time to time...
- Efficient code sequences
- Need to distinguish literals vs. DCs (“constants” vs. “initial values”)
- Conditional assembly
  - Open code, macros may not be easy to convert

(4) Quality of the converted code

- Validity: is it correct, does it give the same results?
  - The single most critical acceptance criterion
- Size
  - Growth (possibly substantial) may cause other problems
- Performance
  - Tuning may be difficult; requires tinkering, knowledge of AL
- Understandability and maintainability
  - A main reason for attempting a conversion
  - May not improve much, and may be (much) worse
- Debuggability
  - Almost always requires knowledge of AL ***and*** HLL!

(4) Rewrite the application in a HLL: summary

- Conversions usually underestimate difficulties, time, people, etc.
  - Extensive project planning, technical preparation
  - Language and data typing differences involve many subtleties
    - data types/structures, value ranges, record layouts, byte order
  - Difficult to find and fix conversion problems
  - Testing rarely uncovers latent errors
- Analyze and document the existing code!
  - ***Always*** a valuable activity, whatever else happens
  - Opportunity to reorganize, modularize, understand the application
- Research recommends *not* converting languages

- What if you don't convert? See Scenario (5) ...

Summary of language conversion issues, scenarios (3) and (4)

- Organization and management concerns
  - Don't expect more than minor improvements; things may be worse
- Staffing and skills
  - Largest burden falls on skilled, experienced staff
  - Dual-language knowledge required
  - Conversion effort typically stressful, especially for key personnel
- Technical issues
  - Typically must rewrite/enhance test suite, and validate it
  - Great volume of tiny details
  - Cannot rely on compilers to “optimize everything”
- Financial factors
  - Potential requirement for hardware upgrades, new software
  - Lost-opportunity costs

(5) Modernize and maintain the application

- Best option may be to modernize without migration
  - Rewriting in a new language adds no net value
- Analyzing/understanding the application (slide 22) always adds value
  - May have done the really useful work already!
- Modern assembler and support tools help considerably (slides 34,35)
- Application enhancements can be done incrementally
  - Statements, code blocks, procedures, modules, data structures
  - Current test suites need no changes (a major benefit!)
- Macro instructions capture repetitive code sequences
  - Structured-programming constructs easy to add; simplify maintenance
  - Reduce the “gap” between abstraction and expression

(5) Modernize and maintain the application ...

- Organization and management concerns
  - Minimal disruption of internal processes, external business activity
  - Existing investments preserved, minimal testing costs
- Staffing and skills
  - Easy-to-learn modern techniques can increase productivity
- Technical issues
  - Improved product quality
  - No degradation of efficiency, code size
  - Modernization updates easy to validate; can do incrementally
- Financial factors
  - No lost-opportunity costs
  - No forced upgrades to hardware and software

(5) Modernize and maintain the application: summary

- Large benefits from “code clean-up”
  - Documentation, modularization immediately beneficial
- Modern tools/techniques improve understandability, maintainability
  - Program analysis and restructuring tools provide useful insights
  - Structured-programming macros can be introduced at minimal cost
  - Application macros reduce coding burdens, increase clarity
    - Business-specific terminology and data types
- Lowest risk and invasiveness of all five scenarios
  - Least impact on staff, processes, end users

High Level Assembler

- ***Many*** enhancements to Assembler Language and its assembler
  - New USING statements enable efficient, readable code
  - New and improved diagnostics
  - New macro/COPY, register, DSECT XREFs; enhanced symbol XREF
  - Many extensions to the conditional-assembly “macro” language
    - Enables rapid creation of efficient application-specific languages
  - ADATA file captures all information about the assembly
    - Supports many other tools and processes
  - Interfaces for I/O exits, external conditional assembly functions
- Faster development, greater productivity, improved code reliability
- Five releases since 1992; Assembler Language is ***not*** dead!
  - Continuous improvements – “It's not your father's assembler!”

High Level Assembler Toolkit Feature

- Enhances productivity by providing six powerful tools:
  1. A flexible **Disassembler**
     - Creates symbolic Assembler Language source from object code
  2. A powerful Source **Cross-Reference Facility**
     - Analyzes code, summarizes symbol and macro use, locates specific tokens
  3. A workstation-based **Program Understanding Tool**
     - Provides graphic displays of control flow within and among programs
  4. A powerful and sophisticated **Interactive Debug Facility** (IDF)
     - Supports a rich set of diagnostic and display facilities and commands
  5. A complete set of **Structured Programming Macros**
     - Do, Do-While, Do-Until, If-Then-Else, Search, Case, Select, etc.
  6. A versatile **File Comparison Utility** (“**Enhanced SuperC**”)
     - Includes special file-search and date-handling capabilities

Review of reasons to change languages

Reasons for changing computer language include (see slide 5)

1. *Skills availability*
   - Internal skills enhancement: easy to motivate, easy to learn
   - Business understanding far more important than HLL knowledge
2. *Standardization on fewer (or one) language(s)*
   - May not be an achievable goal: possible dependence on existing applications, packages, application-specific languages
   - Potential for disturbance of ongoing activities
3. *Maintainability*
   - Maintenance quality depends on programmer quality, not on language
     - Good program structure far more important than programming language
   - Maintenance often more difficult than development
     - Testing rarely exposes all errors
   - Most programmer time spent maintaining products, not programming

Review of reasons to change languages ...

4. *Application portability to a wider range of hardware platforms*
   - Rarely needed for substantial applications
5. *Belief that a new language is “Industrial Strength”*
   - Elaborate facilities may increase learning difficulties
6. *Bulk buying discounts*
   - Typically, for workstation-based development tools
7. *Storage-constraint relief*
   - Updating AL programs is straightforward
8. *Fashion, competition*
   - Advertisements (and academic enthusiasms) aren't always reliable
   - “Everybody's doing it” may not be the best reason to change...

Review of reasons not to change languages

Reasons for ***not*** changing computer language include (see slide 6)

1. *Cost, time, and complexity of conversion effort*
   - Often far greater than anticipated; difficult to manage
     - Language conversion is **not** automatic for “real” programs
   - Distinguish complexity of business logic from its implementation
2. *Quality and maintainability of converted code*
   - Rarely better, often far worse
   - Newly-introduced errors may cause other problems
3. *Continuity of planning, procedures, policies, and expertise*
   - Required changes can impede future development plans
4. *Hardware upgrade costs*
   - Slower, fatter converted code may need new engines
   - People-cost savings may become hardware/software expenses

Review of reasons not to change languages ...

5. *Cost of new software products*
   - Compilers, run-time libraries, debuggers, etc. usually more costly
6. *Loss of flexibility*
   - Less control over code and data layouts
7. *Performance concerns in the new language*
   - HLL and O-O code may be slower, fatter, harder to tune
   - Compilers and runtime libraries for complex languages more likely to be buggy
8. *Loss of function*
   - HLLs may not support necessary capabilities
   - AL can maximize utilization of platform facilities
9. *Re-training costs*
   - Education for new language, modified application
10. *Language longevity:* languages come and go (mostly, go)

Summary observations

- Productivity factors known to work
  - Reliable documentation (how the “business rules” were implemented)
  - Good programmers are by far the industry's best bargain
    - Productivity may differ by factors of 10 to 30
  - Choice of language has little effect on quality, reliability, maintainability
- Organizational and managerial issues
  - Organizations can have 10:1 productivity differences
  - Code reuse can provide large gains; not a technical issue
  - Maintenance costs significantly outweigh development costs
    - Don't assign inexperienced, less-skilled staff to maintenance
  - Measurement (of what?): *may* help learn what helps most
  - Key performance drivers are often sociological, not technical
    - Staff commitment to management “directives” (may blame management for difficulties)

Summary observations ...

- Language change is disruptive
  - Impact often underestimated
  - Once an application is understood, why change its language?
    - Poor algorithms and poor data structures can be written in any language.
- “Changing languages should be avoided if possible”
- There are no “silver bullets”
- Assembler Language is certainly ***not*** the only answer!
  - Need to carefully weigh situational advantages and disadvantages
- Learning Assembler Language (and machine language) benefits understanding all HLLs
  - Even if the chosen HLL is intended to be platform-independent

Assembler Language: what is *actually* bad?

- Requires (some) familiarity with hardware architecture
  - This is an advantage in disguise
- Assembler doesn't enforce program and data structuring
  - Allows great (too much?) flexibility in specifying both
    - Data structuring sometimes embedded in the instruction statements
  - No enforcement of operation-vs.-data type conformance
    - But: very easily implemented with macros
    - And: no invisible data-conversion costs that HLLs may impose
- Has a reputation for obscurity and verbosity
  - Easy to be overwhelmed by details; macros can be enormously helpful
- Early programming techniques less than robust
  - Modern assembler products provide much more help

Assembler Language: what is *good* about it?

- Many helpful facilities and language enhancements
  - Powerful macro facility supports application-specific languages
  - Macro instructions an excellent form of code re-use
- Full access to hardware/software platform services
  - Can support interfaces to and from other languages
- Independent of compiler/run-time release changes
- Full control over instruction sequences
  - Ability to optimize performance, minimize program size
  - Code is correctable without reassembly or re-linking
- Lost source (any language) is recoverable into Assembler Language
- Achieving AL benefits with HLLs may be difficult or costly
  - The strengths of AL are frequently the weaknesses of HLLs

Bibliography: Recommended Reading

1. P. Middleton, **The Costs of Changing to a Fourth Generation Computer Language**, *J. Programming Languages* 2 (1994) pp. 67-76.
2. Andrey A. Terekhov, Chris Verhoef, **The Realities of Language Conversions**, *IEEE Software* Nov./Dec. 2000, pp. 111-124. IEEE Computer Society, Los Alamitos, California.

Other interesting articles:

3. Y.A. Feldman, D.A. Friedman, **Portability by automatic translation: A large-scale case study**, *Artificial Intelligence* **107** (1999) 1-28.
4. Robert L. Glass, **Frequently Forgotten Fundamental Facts about Software Engineering**, *IEEE Software* May/June 2001, pp. 112-111. IEEE Computer Society, Los Alamitos, California.
5. Jeffrey Voas, **Software Quality's Eight Greatest Myths**, *IEEE Software* Sept./Oct. 1999, pp. 118-120. IEEE Computer Society, Los Alamitos, California.