Legacy Software

Dennis Mancl

dmancl@acm.org

Presentation at TCF 2025,
IT Professional Conference track
March 29, 2025

MSWX

MSWX ◊ Mancl ◊
Software ◊ Experts ◊ http://manclswx.com

What do we learn about computer programming?

- Languages
  - Textbooks, tutorials
  - Websites
    - stackoverflow.com
- Algorithms
  - Numeric
  - Searching, sorting
  - Command processing
  - Control
- Tools
  - Code analysis

- Do we learn about extending and fixing code? (on the job…)

What is “legacy”?

Suppose you inherit a table from your grandmother…

Louis Quinze style table

- Great materials (walnut)
- Excellent workmanship
- Strong, beautiful
- Requires some significant maintenance

1950s style table

- Good materials (tubular steel, chrome)
- Solid construction
- Strong, but a bit ugly
- Easy maintenance

Some characteristics of legacy software?

✓ • Code written by someone else, but you still need it
  - You *might* even have some code documentation

? • Written in a "legacy language"
  - COBOL for business applications
  - Fortran for scientific applications
  - C for telecom and PC applications
  - old versions of Visual Basic for user interface applications

- Actually, there is a lot of legacy code in:
  - Java
  - Scripting languages
  - Spreadsheets
  - Javascript

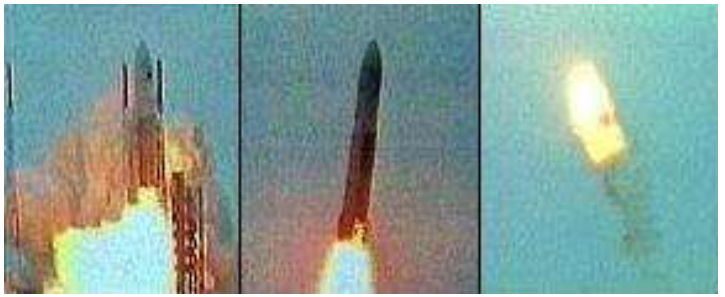
Why is legacy code difficult to work with?

The legacy code "works", but…

- ✓ poor code structure
  - difficult to understand the functions and interfaces
- ✓ inflexible data structures and data types
  - time/date (two-character year, 32-bit time values)
  - fixed character string lengths, fixed array sizes
  - limitations on the range of numeric values, poor handling of negative values (the Ariane 5 case – next slide)
  - using 8-bit ASCII instead of Unicode strings for international applications
- ✓ poor algorithms
  - algorithms that need to be extended or expanded to handle new requirements
- ✓ poor practices in fixing bugs
  - making the code more brittle and trouble-prone

What could go wrong?

- Reusing software seems easy…
  - "This code works fine in Product A, let's reuse it in Product B"
  - Famous disaster – Ariane 5
  - https://jam.dev/blog/famous-bugs-rocket-launch/



They decided to reuse guidance software from Ariane 4

- But Ariane 5 has a much bigger, faster first stage
- Some changes in launch procedures

They forgot to fix the error handling for one "overflow" case

- (overflow converting a 64-bit floating point quantity to a 16 bit integer, triggered a software exception – never needed this case in Ariane 4 launches)
- The error handling code fed incorrect data to another part of the guidance system
- Caused an unnecessary course correction; triggered self-destruct

Why not rewrite everything?

- Rewriting some old software may not be feasible:
  - costly and time-consuming
  - a complete rewrite will still require a lot of testing
  - the original code authors may have understood the problem domain better than the new staff
  - the code contains a lot of useful information about how your business works
  - some customers might rely on some of the existing bugs
- Maybe it is better to work with the old ugly code. But how?

"Scroll down a little..."

We could discuss the economic issues for hours…

Reuse is a shortcut for building new systems

- Time to market is really important
  - Get a working system on the market
  - Use "components" from older legacy software systems
  - Open source components
  - "Smart reuse" is important!

Web apps

- Not just reusing individual buttons and widgets…
- Reuse data access, formatting, and communication code
- Error handling code

Device control

- "Smart home apps"
- Support software – event handling, communication, authentication, encryption
- Reuse of algorithms
- Synchronization and error handling
- Safety monitoring

Legacy code techniques

- wrap and reuse key components
  - Wrappers are extra code that "simplifies" the interface of some legacy modules
- create new "extension points" in the legacy code base
  - Make the code easier to extend
- incremental change; build a "bridge to the new town"
  - Rewrite small parts of the code – code base has both old and new modules side by side
  - If there is a problem in the new code, you can easily reactivate the old code
- selective refactoring - focus on the modules that change frequently

Some notable legacy software

C standard library:
Strings: strcpy, strcat, strcmp
Chars: isspace, isdigit, isalpha
Math: sqrt, log, exp, sin, cos
Memory: malloc, free

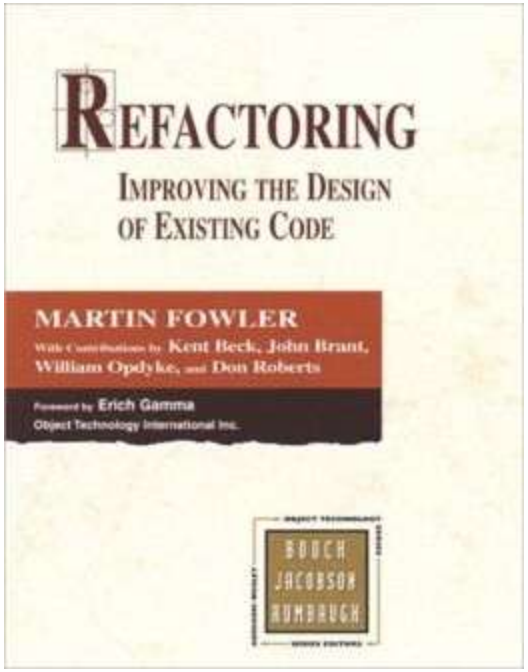
- Standard libraries
  - libc – the "standard C library"
  - OpenSSL – security library used for web software
- Hardware design tools
  - SPICE – design tools for modeling integrated circuit designs
  - Originally a student project at UC Berkeley (1973)
  - Very flexible architecture, relatively easy to extend
- Bitcoin
  - Basic algorithms are pretty old (2009)
  - 2 major Bitcoin Core releases each year (plus minor fix releases)

The rest of this talk

- Refactoring – making small safe changes to improve the code base
- Be conservative (and humble) when you make changes to legacy code
- Your code will someday be "legacy code"
- How do you start to work with a big ugly code base?
- How do you get experience in working on legacy code?

Refactoring

- A *refactoring* is a behavior-preserving program restructuring that can improve the design of software and support evolution and reuse.
- Examples:
  - basic / primitive:
    - renaming variables, functions, and classes
    - Goal: increase **readability**
  - less basic:
    - restructuring existing functions
    - Goal: make individual functions **simpler**; reduce **duplication**
  - more complex:
    - reorganize the modules and classes
    - Goal: set up for new **extensions**



My refactoring talk:
https://youtu.be/-hdvXRfqhdo

Refactoring legacy code

- Each refactoring step is "clean up" work – making the code easier to read and easier to change:
  - a good conservative practice
  - refactoring can be used to clean up and simplify old code
    - Simpler code == less "technical debt"
  - good refactoring requires an organized testing strategy
    - Testing is critical: automated unit tests are very valuable
  - "refactor to understand" is a good way to learn
    - New staff start to understand the structure of legacy modules

Technical debt = the cost of making hasty design decisions… to get the code out on schedule.

- You "owe yourself" some effort to clean up the quick-and-dirty tricks.

Should I use tools to refactor my code?

- Answer: yes and no
  - NO: don't hand over your code to an untrusted tool
  - You may break everything!
  - YES: carefully refactor complex functions with the help of tools
  - But… work in small steps

- Using AI tools…
  - Not too awful for micro-refactoring work
  - But we often use "refactoring to understand" – and tools don't help you learn
  - All AI-written code *must* be inspected by hand… and unit tests are also a good idea

Legacy code is an asset

- Refactoring can be worthwhile, because legacy code can be very valuable
  - "legacy" is the code that is still delivering "value" to the customers
  - don't throw away old code arbitrarily
    - Why not just "rewrite everything from scratch"?
    - Your users depend on the existing functionality…
  - be humble – your predecessors might have known a lot of things that you still haven't learned yet, so don't be too hasty to refactor
  - use a "just enough architecture" approach...

Refactoring to learn

- Use refactoring to explore legacy code:
  - The code is the best documentation of a legacy software system
    - design documents are often out of date or just wrong
    - the code always "tells the truth"
  - But you can't understand everything at once
    - initial exploration: modules, execution paths, communication paths, and databases
    - detailed exploration may include "exploratory refactoring"
    - add in some new "unit tests" to show internal states
    - we make some small changes – new "probes" into the middle of key functions and classes

Learn to add unit tests: "Working Effectively With Legacy Code" by Michael Feathers

What you need to know (1)

- Your really cool application will someday be "legacy code"
  - in the future, someone will complain that the structure of your code is terrible
- You need to be aware of how you design and code
  - a little bit of extra effort on the code structure (understandable names, good separation of concerns, flexible data structures)
- How to slow down the bit rot: Include unit tests in your code base

Take steps to make future evolution easier

What you need to know (2)

- When you are faced with a big chunk of legacy code, don't despair
- Focus on the code that changes most frequently – look at the change history and talk to the maintainers
- Use some legacy code strategies: wrap stable functions, write your own microtests, refactor to understand, build bridges to new functionality
- Keep everything in a source code management system – easier to go back when you make a mistake

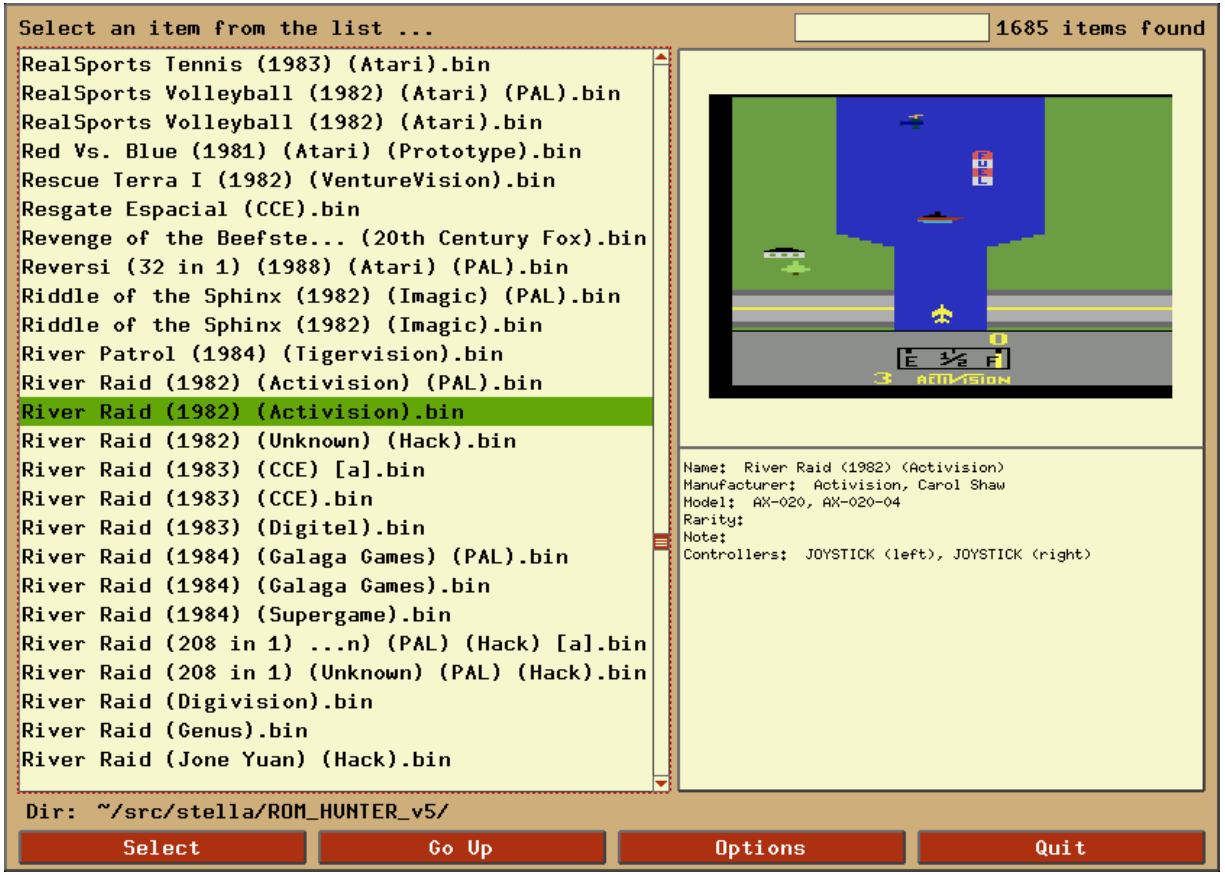
Be organized when you build on top of poor legacy code

Where to get experience

- Volunteer to work on an open source project
- Invest some effort in some of your own old code
  - Don’t do massive rewrites – preserve most of the code structure, but fix inflexible data structures, rename cryptic variable names, and add some unit tests

Emulation: one technique for reusing legacy

- For some old code it is easier to "emulate the original environment" than to port the code to a new environment
  - Example: You want to play some of the old Atari 2600 games
  - Use an Atari emulator to "interpret" the Atari instructions



Stella emulator
http://stella.sourceforge.net

Emulation strategy

- When is emulation a good way to work?
  - Code is really old
  - It is difficult or impossible to get the old hardware
  - Or: flexibility is important
  - You have enough processor power

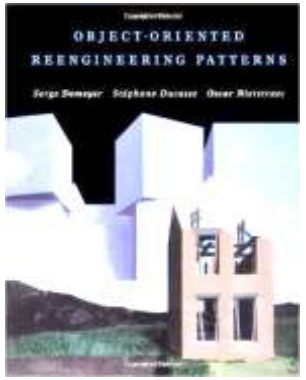
Prediction: We will see a lot of emulation in the "smart home" market:

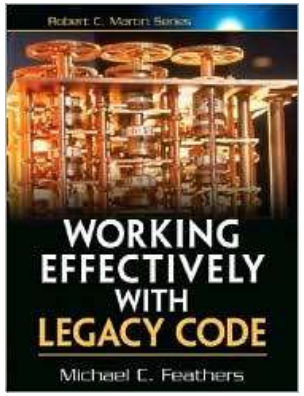
- Appliances and other hardware devices will have hundreds or thousands of "versions"
- Increased flexibility – system integrators can write software to interact with multiple versions
- "Microservices architecture" is a practical option – collaboration through simple messages

https://martinfowler.com/articles/microservices.html

To learn more

- Object-oriented Reengineering Patterns
  - This book gives some good advice on how to start to read and refactor legacy code
  - This book is "open source" -- https://scg.unibe.ch/assets/download/oorp
- Working Effectively With Legacy Code
  - Good explanations of how to introduce unit tests into your legacy code modules
- Clean Code
  - Some advice on how to write code that is easier to maintain
  - Readability, good function names and variable names, short functions





To learn more

- My "Refactoring Project" –
  - https://manclswx.com/projects/refactoring.html
  - Guidelines about refactoring
  - A short essay about legacy code issues
  - More books, articles, tutorials about dealing with old code