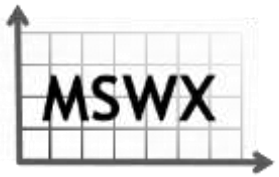
Legacy Software:

Old Software that Hasn't Been Forgotten

Dennis Mancl

dmancl@acm.org



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

Legacy software: a trip to the past

In this talk, we will discuss legacy software and software engineering practices

- Software development = the nuts and bolts of building software products
- Software engineering = developing applications at a reasonable cost with good quality

A Legacy System is an old software system that is still in use.

- It is still providing value to users.
- In theory, we could rewrite the system.
- In practice, customers might be happier if we just make updates to the existing system.

Rational choices

- It is getting too expensive to keep the old system working
  - Written in a “legacy language” (Cobol, Fortran, C)
  - Difficult to newly-hired developers to work with the old design
  - We are spending a lot of time fixing old bugs
  - Customers want a more modern user interface
- But there is some value in the old code
  - The original developers really knew a lot about the problem domain
  - A replacement system might not be able to implement some of the complex logic or business rules

Need to make a calm, rational decision about replacement

Thinking about the changes we make to software

- I used to work on telecom software… every “product release” has a combination of:
  - Old features that *must* work
  - New features that the customers are clamoring for…

Managers and customers say… don’t break the old stuff, but we want the new stuff to be great!

Why software changes

- Wait a minute… Software doesn't wear out! Why do we ever need to change it?
  - Bugs / errors / defects
  - Adding a few new features… new requirements
  - Porting to work with new devices / new version of operating system
- 3 types of changes
  - corrective
  - perfective
  - adaptive

[B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, 1980]

A software system is always changing and improving, unless it is dead and buried!

Discovery costs: the root of evolution issues

- What are “discovery costs”?
- When you pick up a small software system, what questions do you ask?
  - How do I “build” the software? (create an executable image)
  - What environment do I need to run it? Are there some other software packages that need to be installed first?
  - What do I need to read first if I need to make changes or adjustments? (Reconfiguring the system to work in a different environment – timezone – language – network configuration)
  - How can I test that everything really works as advertised?

What are discovery costs?
“The things you need to learn to do the job you need to do”

Discovery costs are everywhere

- When you hire new staff:
  - They don’t know your build process
  - They don’t know the unit test framework
  - They don’t know the range of customer configurations
  - They don’t know the long history of how the current product evolved…
- So, how effective is a new employee on Day 1?
  - Absolutely useless?
  - Not quite, but close…
  - There will be a huge learning curve… but also an “opportunity”

“New staff members ask dumb questions”
(but some are not so dumb!)

Discovery costs – not just new employees

- When you transfer staff to a different project (or a different job role):
  - They don't know that project's build process
  - They don't know that project does its testing
  - They don't know much about that project's customers
  - They don't know that project's long history…
- So, how effective is a transferred employee on Day 1?
  - Not as bad as a new employee…
  - But it can be an expensive mistake to keep churning staff around!

Rule of thumb (from Bell Labs study):
It takes "three release cycles to get up to speed"
(to reduce discovery costs below 40%)

Learning activities

- What to do – to mitigate potential discovery issues:
- Activities to make learning about the existing design and code as easy as possible
  - **code reading**
    - make sure that developers have time (and tools) to explore the existing code base
  - **code reviews** and **internal chalk talks**
    - experienced members of the development team talk to others about the design of key components
  - add missing **documentation**
    - “reverse engineering” the missing design documents for legacy modules and subsystems
  - **simple models**
    - building a simple software model
    - use informal brainstorming-like methods such as CRC cards,
    - if you use more formal models (such as state machines and UML diagrams), keep them simple
  - running small **experiments** in the lab - building small simplified examples that reuse parts of existing components in a new context

Why invest in code reading?

- Suppose that staff members are not given sufficient time to understand/discover the code for which they are responsible
  - The most common approach for adding new functionality = “clone-and-modify” code
  - Also called “copy-and-paste reuse“
  - This is the worst kind of reuse, because the resulting code becomes more complex and more brittle – and repair costs will be higher.

  - Suppose the original code always seemed to work fine
  - But the original code will not work correctly when connecting to a new remote database…
  - If that original code is “copied 5 times” in a series of subsequent releases... developers and testers will need to track down the defect 5 times!

Why working with legacy code is hard

Poor code, poor design

- 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
  - 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

Refactoring process – improving a little at a time

The refactoring process involves:

- making small changes that improve the code's structure
- but the changes do not change the functionality in the code
- and it should be easy to test that the code still works

We are "cleaning up the code"

- Making it easier for us (and other people) to work with
- Easier to fix errors and add new functionality

The first "refactoring tools" started to appear in 1990.

William F. Opdyke and Ralph E. Johnson (September 1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems."

Can we make changes safely?

Refactoring process

The refactoring process is “focused”

- Limited refactoring … only in the parts of the code base where you plan to make changes

We “clean up” our mistakes

- Improve the internal structure

We also try to cut down on duplicated code

- Don’t go crazy… but it will help in the long run
- Reducing discovery costs for the next set of developers

Code smells

Refactoring based on “code smells”

- Every code smell is a place in the code where the original developer wasn’t thinking about keeping the code “clean”

➢ Focus your refactoring based on “things that smell bad”

➢ The refactoring process is driven by the “intuition” of the developers… only you can answer these questions:

When is a function too big?

When does a function have too many arguments?

When is a Boolean conditional expression too complex?

How much duplicated code is OK?

Which comments are too confusing?

Where do I need more unit tests?

Instead of asking “Where are we making lots of changes?” – Agile developers consider Code Smells

It’s up to you to decide!

Don’t refactor something just because the “experts” say that you *must* follow their coding rules…

Other code improvements

There are more ambitious things you can do with refactoring...

My favorite... replace custom code with calls to standard library functions and classes

- particularly useful in languages with rich libraries - Java and Python
- the use of standard libraries can shrink the total code size, increase understandability, and sometimes improve performance
- can this be taken too far? absolutely!

It is also useful to refactor to “add more tests”…

Other code improvements

Refactoring to add unit tests to a legacy system without unit tests

- there is a process to find or create “seams” in the code
- places where you can add a simple unit test to exercise one or more functions in the code
- best explanation: “Working Effectively with Legacy Code” by Michael Feathers
- Michael says “get that old code under unit-test control”

Refactoring to “learn”

Use refactoring to explore legacy code:

The code is the best documentation of a legacy software system -- much better than any design documents.

- 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 should focus on understanding the modules, execution paths, communication paths, and databases
- detailed exploration may include “exploratory refactoring”
- we make some small changes -- new “probes” into the middle of key functions and classes

The goal is to “learn” – and the refactoring work may actually be thrown away after you understand the legacy system!

Book: Object Oriented Reengineering Patterns by Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz

Free download!!!
http://scg.unibe.ch/assets/download/oorp

Chapter titles-

- First Contact Patterns
- Initial Understanding
- Detailed Model Capture
- Tests
- Migration Strategies
- Duplicated Code
- Redistribute Responsibilities
- Introduce Polymorphism

Summary

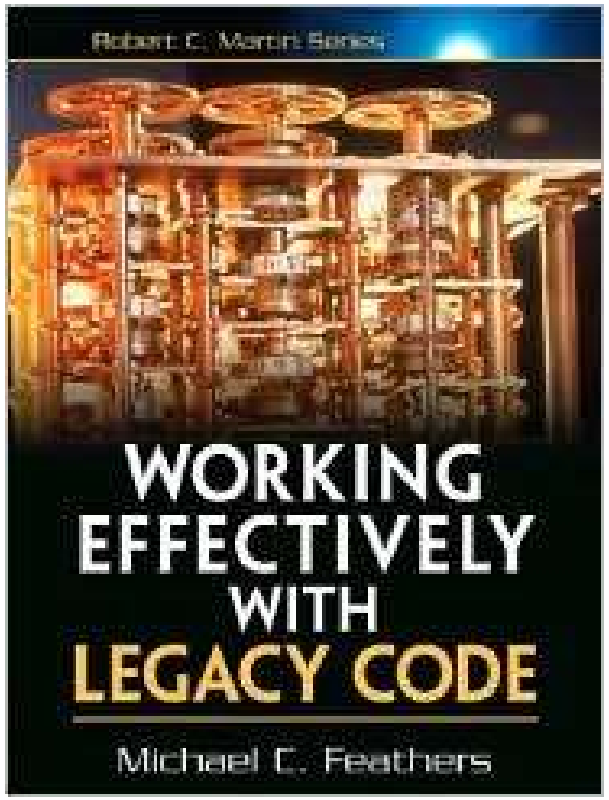
Legacy code isn't easy, but it is worthwhile learning about

- Learn some legacy software reengineering techniques
- Watch out for "discovery costs"
- Add unit tests to your legacy code – to support safe refactoring

Goal: better quality, easier to make code changes

More resources:
manclswx.com/projects/refactoring.html

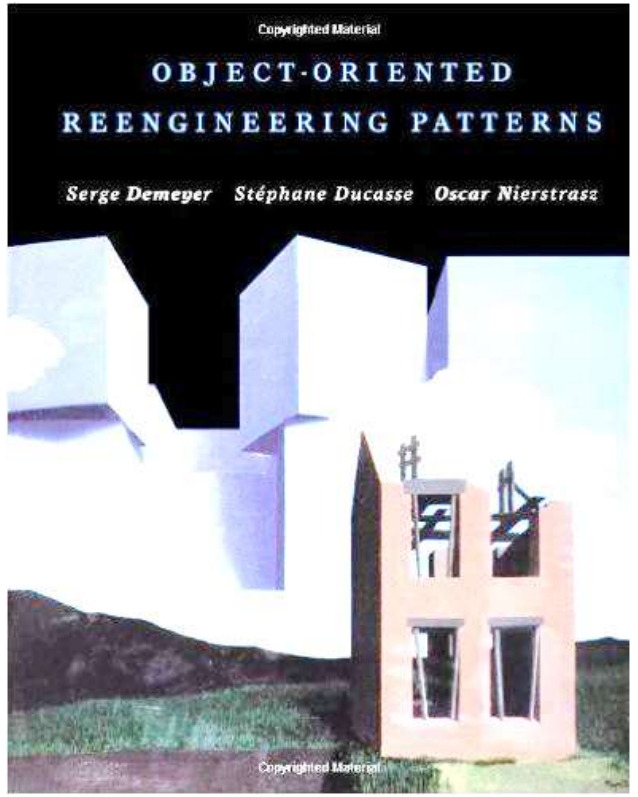
Two good books



Michael Feathers explains how to introduce unit tests into your legacy code modules.

- He believes that automated tests are the best way to get legacy code under control

This book is “open source” -- https://scg.unibe.ch/assets/download/oorp



Reading code is hard because no one knows where to start.

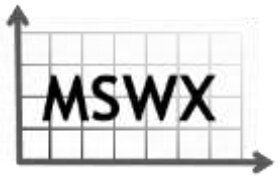
- This book gives some good advice on how to start to read and refactor legacy code

Questions?

More resources:
manclswx.com/projects/refactoring.html

Dennis Mancl

dmancl@acm.org



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