Legacy code – how to work effectively with old software

Dennis Mancl, MSWX Software Experts

There are a number of important design and implementation issues for anyone in the industrial software development – because “legacy code” is important to all of us.

Legacy code is any old code that is still providing value to its users. Although many software developers would rather work on new products and new technologies, the customers usually prefer evolutionary change. When we add new features to an existing and popular product, it is easier for our customers to incorporate that new release into their workflow – much easier than changing their business processes to fit a completely new product.

It is a big challenge for developers to build on top of existing software products. Why? Because when a developer has to work with someone else's code, the code may depend on some undocumented design decisions that are not always obvious. Cryptic code, clever design tricks, and unexpected side effects are the biggest obstacle to software evolution. Documentation is rarely much help: design documentation and code comments are not always accurate, and it is not uncommon to have code with no documentation at all.

Developers need to follow a few common rules when working on legacy systems:

- Adding new functionality should not break any old functionality.
- Internal architecture and design changes are occasionally needed to implement new functionality, but you must be careful not to degrade the performance of common user scenarios.
- Code readability is important – and keeping the code consistent is an important part of readability. If you make changes within a legacy module, you should follow the code formatting style of the module (even if the code doesn't follow your organization's standard formatting style).
- The most important “clean code” rule is to reduce duplicate code. There are far too many code bugs that made worse by “copied code” – “I copied this code into my code from component A because it worked there...” When someone later fixes a bug in component A, they will usually miss making a similar fix to the copy.
- Use classes to package some of the legacy code, even if it is written in C or assembler. If you are extending an existing module, consider creating a “wrapper” around the old code. The wrapper can encapsulate the old basic behavior, and the new functionality can be implemented in extension classes or collaborating classes.
- Don't rewrite everything. You might think you understand an existing code module – and you think you can do it better. But you might not have as much domain knowledge as the original code authors – and your “faster version” might be missing some critical scenarios. It is better to "be humble" – to do more selective refactoring instead of a complete rewrite.
- Add unit tests to the legacy code. There are two benefits. Unit tests are “executable documentation” that helps everyone learn about how the code works. Unit tests also make the code easier to refactor safely.
- If a software product doesn't have adequate requirements documentation, consider creating a set of “use cases” (basic scenarios and failure scenarios) to outline the external interactions with the code – this is a “discovery” activity that makes reengineering and refactoring easier.
- Be aware of the long-term cost of Technical Debt. It is OK to make some quick and dirty design decisions when extending a legacy system. When you make a rapid change to an existing system, it is an experiment – an attempt to build some new features to check that they have positive value for the users. But these quick changes need to be “refactored” as soon as possible, in order to reduce the cost of future changes. If the technical debt in the code base becomes too large, the system will “die” because no one will be able to afford to add new features.

There are six main legacy code issues:

- **Topic 1: Co-evolution** – or "why is software change inevitable?" A new system will trigger behavior changes among the users of a system, so there will be new requirements to react to those changes.
- **Topic 2: How to "plan ahead"** for software change – there are many standard techniques for flexible data structures and flexible classes in a software design, which can reduce the time and effort to react to change.
- **Topic 3: Discovery costs** – also known as "learning curve" – the costs of learning what you need to learn in order to do the job you must do.
- **Topic 4: Wrapper techniques** – how to reduce discovery costs for most developers by creating solid software components from parts of a legacy code base.
- **Topic 5: Refactoring** – how to improve the structure of a software application or library, while keeping the functionality unchanged. Refactoring is a fundamental tool for reducing discovery costs.
- **Topic 6: Human issues** in long-term software development – investing in the skills and knowledge of a "team" of developers, so that legacy code can have a long life. The most critical knowledge is "problem domain knowledge," and this kind of knowledge is usually undervalued.

Note that topics 1, 3, and 6 are not purely technical software design and coding topics – they are the shared responsibility of developers and managers.

Topics 2, 4, and 5 are technical practices – mostly focused on developer activities. Every developer can learn to use these practices effectively. They are not limited to senior developers.

Both Co-evolution and Discovery Costs are unfamiliar terms to most software professionals today.

Co-evolution

Co-evolution is the biggest driver of software change. If a new software system is successful, it will need to be updated frequently, because the users will find new uses – the software will need to grow to handle new usage scenarios.

Although most software evolution is unpredictable, there are a number of well-known design issues that create unnecessary obstacles:

- Fixed-sized data structures and arrays (the software needs to be edited and recompiled to handle larger inputs)
- Strings that are restricted to standard 7-bit ASCII (better to support international character sets such as Unicode)
- Interoperability restrictions – software that is designed to run only as a "main program" in a standalone environment with fixed locations for file, database, and network resources (instead of "execute as a re-entrant component with configurable resources")

Discovery costs

Discovery Costs are the learning costs that are often ignored in the planning process for software. Most of the software estimation models say – if you have M people developing a software system that will be approximately N lines of code, it will be a relatively *linear* process. The development time should be K * N / M, where K is an average developer productivity in "lines of code per day." In other words, the developers will design and code at a uniform rate for the entire development interval – with a little bit of communications overhead for module interfaces and testing.

But for legacy code, there is usually a large learning curve – the developers need to understand the structure of the legacy system, the design assumptions, the standard libraries, and the logging and error handling policies. If the system has good and consistent design documentation, if the code is well-structured, the code comments are simple and accurate, and the code isn't too complicated, the learning curve might be relatively low. But most legacy code is not well-structured or well-documented. The code might have been pretty good initially, but over the years, a combination of bug fixes and hastily-added new features have added design inconsistencies. The code comments are often incorrect or misleading, because no one has taken the time to update them as the design and code has evolved.

There are two big issues related to Discovery Costs:

- How can you make a reliable development estimate given that the discovery costs are going to depend on the level of developer experience and the level of complexity in the legacy code base?
- How can we keep the complexity from getting worse – at a reasonable cost?

One of the best ways to deal with the uncertainties caused by discovery costs: use iterative and incremental development. Most of the industry data shows that the discovery costs are relatively high for developers who are working on their first and second releases of a legacy product. If the release interval is once per year, many of the new developer will be working at a lower level of productivity for a long time. When the work is done in small cycles, with integration, testing, and some “internal delivery” activities in each cycle, the developers will gain experience faster, and the discovery cost curve will shrink sooner.

There are other important “discovery cost management” practices:

- code reading
- internal chalk talks by members of the development team about the design of key components
- “reverse engineering” the missing design documents for legacy modules and subsystems
- building a simple software model (using informal brainstorming-like methods such as CRC cards, or using more formal models such as state machines and UML diagrams)
- running small experiments in the lab - building small simplified examples that reuse parts of existing components in a new context

There are some good lessons to learn from “open source” projects. Most open source software is built by volunteers, so it would seem that “discovery costs” shouldn't be a big economic factor. But – if it is difficult for a volunteer to get started on an open source product because of its complexity and long learning curve, there will be fewer volunteers willing to expend the effort. So the most successful open source projects spend a lot of time on accurate design documentation, keeping the code base clean, and writing and running automated unit tests. Also, most open source products are developed using very simple languages and tools – C is more popular than C++ in the open source world, and Python is now very popular for new projects. By using simple tools and environments, the cost of getting started in an open source project can be kept low.

Human issues

One of the worst possible management strategies for working with legacy code is – “we will move people around between subsystems and job responsibilities every six months.” Why is this bad? Developers will always be deep into the learning curve, each time they are moved from one area to another. It is better to “value your experts” – and allow them time to train the next round of experts over a period of months and years.

Of course, experts need to keep growing. We often ignore the “potential” of our experts when there is a flurry of new feature requests or a shower of field defects. Managers are tempted to keep the focus on the short-term issues: build new features quick and dirty, fix the bugs as fast as possible. This narrow focus on the immediate work assignment is not good for the future. Expert developers need some time to

reflect, to “fix the root cause of the problem” instead of the immediate bug.

Software development is a very “information intensive” job – and developers need to keep learning every day. If we turn the development job into a high-pressure effort to write the maximum amount of code, the quality (and the “real” productivity) will suffer. The bottom line: If staff members are not given sufficient time to understand/discover the code for which they are responsible, their lack of knowledge will cause more “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.

Existing literature on legacy code issues

Several books have useful advice and good practical techniques for making legacy code easier to work with:

- Object-Oriented Reengineering Patterns - Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz
- Working Effectively With Legacy Code - Michael Feathers
- Clean Code - Robert Martin
- Refactoring to Patterns - Joshua Kerievsky

For more information

This essay is part of the “Refactoring Project” that can be found at:

https://manclswx.com/projects/refactoring.html

This project explores the processes that software developers use to restructure and improve existing code – with the notion that “reusing” is often much better than building a completely new software system.