Understanding the Problems with Legacy Systems

SADAF FARHAN

Defining a Legacy Project

There's always hope for revitalizing a legacy project, no matter how far gone it may first appear.

"Any existing project that's difficult to maintain or extend".

Defining a Legacy Project

A project encompasses many other aspects, including:

- Dependencies on other systems
- The infrastructure on which the software runs
- Project documentation
- Methods of communication, such as between developers, or between developers and stakeholders

Of course, the code itself is important, but all of these factors can contribute to the quality and maintainability of a project

Characteristics of Legacy Projects

Characteristics of Legacy Projects

Old

Usually a project needs to exist for a few years before it gains enough entropy to become really difficult to maintain.

In that time, it will also go through a number of generations of maintainers.

With each of these handoffs, knowledge about the original design of the system and the intentions of the previous maintainer is also lost

Characteristics of Legacy Projects

Large

It goes without saying that the larger the project is, the more difficult it is to maintain.

There is more code to understand, a larger number of existing bugs

The size of a project also affects decisions about how the project is maintained.

Large projects are difficult and risky to replace, so they are more likely to live on and become legacy

Characteristics of Legacy Projects

Inherited

Legacy projects are usually inherited from a previous developer or team.

The people who originally wrote the code and those who now maintain it are not the same individuals.

May even be separated by several intermediate generations of developers.

This means that the current maintainers have no way of knowing why the code works the way it does, and they're often forced to guess the intentions and tacit design assumptions of the people who wrote it.

Characteristics of Legacy Projects

Poorly Documented

Keeping accurate and thorough documentation is essential to long-term survival of legacy projects.

Unfortunately, if there's one thing that developers enjoy less than writing documentation, it's keeping that documentation up to date.

Characteristics of Legacy Projects

```
/**
 * Retrieve a list of summaries of the most popular threads.
 *
 * @param numThreads
 *      how many threads to retrieve
 * @param recentMessagesPerThread
 *      how many recent messages to include in thread summary
 *      (set this to 0 if you don't need recent messages)
 * @return thread summaries in decreasing order of popularity
 */
public List<ThreadSummary> getPopularThreads(
        int numThreads, int recentMessagesPerThread);
```

Exceptions to the Rule

Linux kernel:

It’s been in development since 1991, so it’s definitely old, and it’s also large having around 15 million loc at the time of writing.

Even then Linux kernel has managed to maintain a very high level of quality.

A static analysis scan on the kernel revealed it to have a defect density of 0.66 defects/kloc which is lower than many commercial projects of a comparable size

Some guesses … why?

Exceptions to the Rule

Linux continues to be a ‘model citizen’ **open source project** for good software quality.

Primary reason for Linux’s continued success as a software project is its culture of open and frank communication.

All incoming changes are thoroughly reviewed, which increases information-sharing between developers.

Reviews of Linux Kernel Maintainer

Well, it finds bugs. It improves the quality of the code. Sometimes it prevents really really bad things from getting into the product. Such as rootholes in the core kernel. I've spotted a decent number of these at review time.

It also increases the number of people who have an understanding of the new code—both the reviewer(s) and those who closely followed the review are now better able to support that code.

Also, I expect that the prospect of receiving a close review will keep the originators on their toes—make them take more care over their work.

—Andrew Morton,
kernel maintainer, discussing the value of code review in an interview with LWN in 2008

Legacy Code

Legacy Code

Common characteristics seen in legacy code include:

Untested, untestable code.

Inflexible code.

Code encumbered by technical debt.

Untested, untestable Code

Technical documentation for software projects is usually either nonexistent or unreliable, tests are often the best place to look for clues about the system's behavior and design assumptions.

A good test suite can function as the de facto documentation for a project.

Tests can even be more useful than documentation, because they're more likely to be kept in sync with the actual behavior of the system.

Ethically responsible developer will take care to fix any tests that were broken by their changes to production code.

Listing 1.1 Some untestable code

```
public class ImageResizer {
    /* Where to store resized images */
    public static final String CACHE_DIR = "/var/data";

    /* Maximum width of resized images */
    private final int maxWidth =
            Integer.parseInt(System.getProperty("Resizer.maxWidth", "1000"));

    /* Helper to download an image from a URL */
    private final Downloader downloader = new HttpDownloader();

    /* Cache in which to store resized images */
    private final ImageCache cache = new FileImageCache(CACHE_DIR);

    /**
     * Retrieve the image at the given URL
     * and resize it to the given dimensions.
     */
    public Image getImage(String url, int width, int height) {
        String cacheKey = url + "_" + width + "_" + height;

        // First look in the cache
        Image cached = cache.get(cacheKey);
        if (cached != null) {
            // Cache hit
            return cached;
        } else {
           // Cache miss. Download the image, resize it and cache the result.
            byte[] original = downloader.get(url);
            Image resized = resize(original, width, height);
            cache.put(cacheKey, resized);
            return resized;
        }
    }

    private Image resize(byte[] original, int width, int height) {
        ...
    }
```

Difficult to Test

Implementations of its dependencies (the downloader and the cache) are hardcoded.

Ideally we would like to mock these in tests to avoid actually downloading files from the internet or storing them on the filesystem.

Provide a mock downloader that simply returns some predefined data when asked to retrieve an image from http://example.com/foo.jpg.

But the implementations are fixed and we have no way of overriding them for your tests.

Difficult to Test

We are stuck with using the file-based cache implementation

But can we at least set the cache's data directory so that tests use a different directory from production code?

Nope, that's hardcoded as well.

Difficult to Test

The ***maxWidth*** field is set via system property so you can change the value of this field and test that the image-resizing logic correctly limits the width of images.

But setting system properties for a test is extremely cumbersome. You have to:

- Save any existing value of the system property.
- Set the system property to the value you want.
- Run the test.
- Restore the system property to the value that you saved. You must make sure to do this even if the test fails or throws an exception.

You also need to be careful when running tests in parallel, as changing a system property may affect the result of another test running simultaneously.

Inflexible Code

A common problem with legacy code is that implementing new features or changes to existing behavior is inordinately difficult.

What seems like a minor change can involve editing code in a lot of places.

To make matters worse, each one of those edits also needs to be tested, often manually.

Understanding Code Inflexibility

Suppose an application defines two types of users: Admins and Normal users. Admins are allowed to do anything, whereas the actions of Normal users are restricted.

The authorization checks are implemented simply as **if statements**, spread throughout the codebase.

It’s a large and complex application, so there are a few hundred of these checks in total.

```
public void deleteWibble(Wibble wibble)
                    throws NotAuthorizedException {
    if (!loggedInUser.isAdmin()) {
        throw new NotAuthorizedException(
            "Only Admins are allowed to delete wibbles");
    }
        ...
```

Understanding Code Inflexibility

One day you're asked to add a new user type called Power User.

These users can do more than Normal users but are not as powerful as Admins.

So for every action that Power Users are allowed to perform, you'll have to search through the codebase, find the corresponding if statement.

```
public void deleteWibble(Wibble wibble)
                    throws NotAuthorizedException {
    if (!(loggedInUser.isAdmin() || loggedInUser.isPowerUser()) {
        throw new NotAuthorizedException(
            "Only Admins and Power Users are allowed to delete wibbles");
    }
        ...
}
```

Code Encumbered by Technical Debt

It’s often more useful and appropriate to ship something that works than to spend excessive amounts of time striving for a paragon of algorithmic excellence.

But every time you add one of these good enough solutions to your project, you should plan to revisit the code and clean it up when you have more time to spend on it.

Every temporary or hacky solution reduces the overall quality of the project and makes future work more difficult.

If you let too many of them accumulate, eventually progress on the project will grind to a halt

Code Encumbered by Technical Debt - Example

Imagine your company runs InstaHedgehog.com, a social network in which users can upload pictures of their pet hedgehogs and send messages to each other about hedgehog maintenance.

The original developers didn’t have scalability in mind when they wrote the software, as they only expected to support a few thousand users.

Specifically, the database in which users’ messages are stored was designed to be easy to write queries against, rather than to achieve optimal performance.

Code Encumbered by Technical Debt - Example

At first, everything ran smoothly, but one day a celebrity hedgehog owner joined the site, and InstaHedgehog.com's popularity exploded!

Within a few months, the site's userbase had grown from a few thousand users to almost a million.

The DB, which wasn't designed for this kind of load, started to struggle and the site's performance suffered.

The developers knew that they needed to work on improving scalability, but achieving a truly scalable system would involve major architectural changes, including sharding the DB and perhaps even switching from the traditional relational DB to a NoSQL datastore

Code Encumbered by Technical Debt - Example

In the meantime, all these new users brought with them new feature requests.

The team decided to focus initially on adding new features, while also implementing a few stop-gap measures to improve performance.

This included introducing ad hoc caching measures wherever possible, and throwing hardware at the problem by upgrading the DB server.

Unfortunately, the new features vastly increased the complexity of the system, partially because their implementation involved working around the fundamental architectural problems with the DB.

The caching systems also increased complexity, as anybody implementing a new feature now had to consider the effect on the various caches. This led to a variety of obscure bugs and memory leaks

Legacy Infrastructure

Legacy Infrastructure

Just looking at the code doesn’t show you the whole picture.

Most software depends on an assortment of tools and infrastructure in order to run, and the quality of these tools can also have a dramatic effect on a team’s productivity.

Development environment.

Outdated dependencies.

Heterogeneous environments.

Development Environment

How much time it takes to first checking the code to do the following?

- View and edit the code in your IDE
- Run the unit and integration tests
- Run the application on your local machine

It might take several days for Legacy Project

Downloading, installing, and learning how to run whatever arcane build tool the project uses

Running the mysterious and unmaintained scripts you found in the project's /bin folder

Taking a large number of manual steps, listed on an invariably out-of-date wiki page

Development Environment Matters

There are good reasons to make the project setup procedure as smooth as possible.

First, it’s not just you who has to perform this setup. Every single developer on your team, now and in the future, will have to do it as well.

Second, the easier it is to set up the development environment, the more people you can persuade to contribute to the project. When it comes to software quality, the more eyes you have on code the better.

Outdated Dependencies

Nearly every software project has some dependencies on third-party software. For example, if it's a Java servlet web application, it will depend on:

- Java.
- Needs to run inside a servlet container such as Tomcat.
- May also use a web server such as Apache.
- Might also use various Java libraries such as Apache Commons.

The rate at which these external dependencies change is outside of your control. Keeping up with the latest versions of all dependencies is a constant effort, but it's usually worthwhile.

Heterogeneous Environments

Mostly software will run in a number of environments during its lifetime.

- Developers run the software on their local machines.
- They deploy it to a test environment for automatic and manual testing.
- It is deployed to a staging environment.
- It is released and deployed to the production environment.

The point of this multistage process is to verify that the software works correctly before releasing it to production, but the value of this verification is directly affected by the degree of parity you can achieve between the environments.

Reasons for Divergence in Heterogeneous Environments

Upgrading trickle down from production.

Different tools in different environments.

Adhoc changes.

Production-only bugs.

Legacy Culture

Legacy Culture

The term legacy culture is perhaps a little contentious—nobody wants to think of themselves and their culture as legacy

Fear of change

Knowledge Silos

Fear of Change

Which features are no longer in use and are thus safe to delete?

Which bugs are safe to fix? (Some users of the software may be depending on the bug and treating it as a feature.)

Which users need to be consulted before making changes to behavior?

Because of this lack of information, many teams come to respect the status quo as the safest option, and become fearful of making any unnecessary changes to the software.

Fear of Change

If the team can maintain a sense of perspective, weighing each change's risks against its benefits, as well as actively seeking out the missing information that will help them to make such decisions, the software will be able to evolve and adapt to change.

Table 1.1 Changes, benefits and risks for a legacy project

Changes	Benefits	Risks	Actions required
Removing an old feature	• Easier development • Better performance	• Somebody is still using the feature	• Check access logs • Ask users
Refactoring	• Easier development	• Accidental regression	• Code review • Testing
Upgrading a library	• Bug fixes • Performance improvements	• Regression due to a change in the library's behavior	• Read the changelog • Review library code • Manually test major features

Knowledge Silos

The biggest problem encountered by developers when writing and maintaining software is often a lack of knowledge.

Domain information about the users' requirements and the functional specifications of the software

Project-specific technical information about the software's design, architecture, and internals

General technical knowledge such as efficient algorithms, advanced language features, handy coding tricks, and useful libraries

Knowledge Silos

Factors contributing to a paucity of communication within a team can include

Lack of face to face communication.

Code ego

Busy face

The End