

Software Re-Engineering

By: Usman Ghani

Lecturer

Computer Science

1.1 Legacy project



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



Note that we're talking about a project here, not just a codebase.



Tend to focus on the code.

1.1.1 Characteristics of legacy projects

Few features that many legacy projects have in common

- Old
- Large
- INHERITED
- POORLY DOCUMENTED

1.1.2 Exceptions to the rule

- Just because a project fulfills some of the preceding criteria doesn't necessarily mean it should be treated as a legacy project.
- A perfect example of this is the Linux kernel. It's been in development since 1991, so it's definitely old, and it's also large.

<http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf>.

1.2 Legacy code

- The most important part of any software project, especially for an engineer, is the code itself.

1.2.1 Untested, untestable code

- A 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. In fact, 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.
- A socially responsible developer will take care to fix any tests that were broken by their changes to production code.

System Properties for Test

- 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.

1.2.2 Inflexible code

- A common problem with legacy code is that implementing new features or changes to existing behavior is inordinately difficult.
- To make matters worse, each one of those edits also needs to be tested, often manually.

1.2.3 Code encumbered by technical debt

- Every developer is occasionally guilty of writing code that they know isn't perfect but is good enough for now.
- In fact, this is often the correct approach.
- As Voltaire wrote, le mieux est l'ennemi du bien (perfect is the enemy of good).

1.3 Legacy infrastructure



QUALITY OF THE CODE IS A MAJOR FACTOR IN THE MAINTAINABILITY OF ANY LEGACY PROJECT.



MOST SOFTWARE DEPENDS ON AN ASSORTMENT OF TOOLS AND INFRASTRUCTURE IN ORDER TO RUN.



THE QUALITY OF THESE TOOLS CAN ALSO HAVE A DRAMATIC EFFECT ON A TEAM'S PRODUCTIVITY.

1.3.1 Development environment

1

About the last time you took an existing project and set it up on your development machine.

2

View and edit the code in your IDE

3

Run the unit and integration tests

4

Run the application on your local machine

1.3.1 Development environment

01

Setting up a legacy project often involves a combination of

02

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

03

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

04

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

1.3.2 Outdated dependencies



NEARLY ANY SOFTWARE PROJECT HAS SOME DEPENDENCIES ON THIRD-PARTY SOFTWARE.



FOR EXAMPLE, IF IT'S A JAVA SERVLET WEB APPLICATION, IT WILL DEPEND ON JAVA. IT ALSO NEEDS TO RUN INSIDE A SERVLET CONTAINER SUCH AS TOMCAT AND MAY ALSO USE A WEB SERVER SUCH AS APACHE.



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.

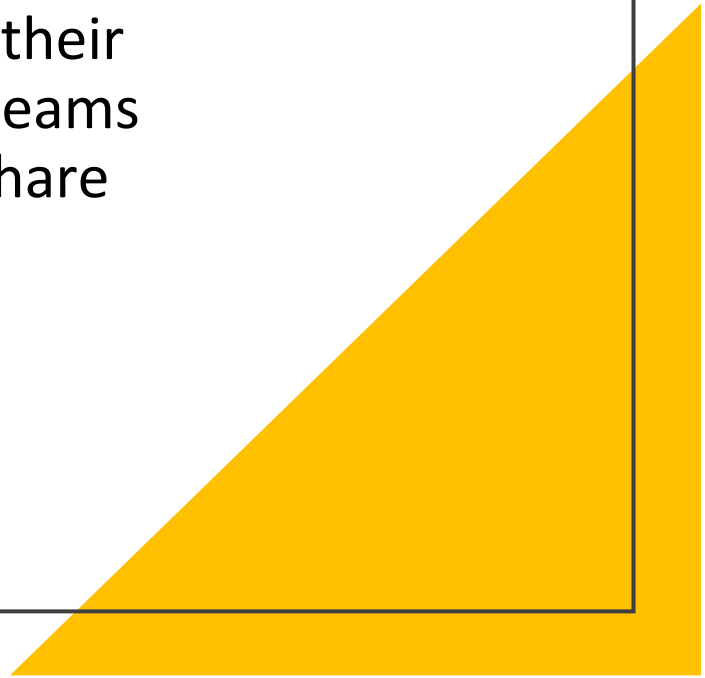
1.3.3 Heterogeneous environments

Most software will be run in a number of environments during its lifetime. The number and names of the environments are not set in stone, but the process usually goes something like this:

1. Developers run the software on their local machines.
2. They deploy it to a test environment for automatic and manual testing.
3. It is deployed to a staging environment that mimics production as closely as possible.
4. It is released and deployed to the production environment (or, in the case of packaged software, it's shipped to the customer)

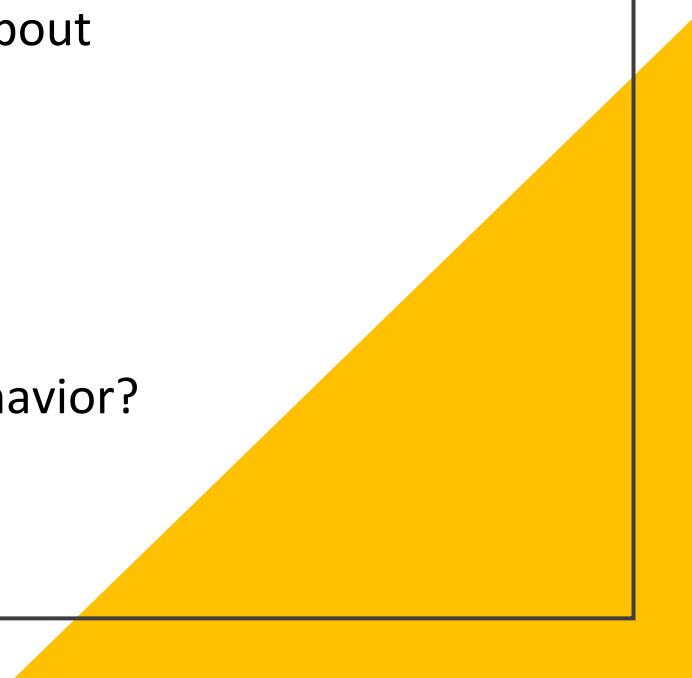
1.4 Legacy culture

- The term legacy culture is perhaps a little contentious—nobody wants to think of themselves and their culture as legacy—but I've noticed that many software teams who spend too much time maintaining legacy projects share certain characteristics with respect to how they develop software and communicate among themselves.



1.4.1 Fear of change

Many legacy projects are so complex and poorly documented that even the teams charged with maintaining them don't understand everything about them

1. Which features are no longer in use and are thus safe to delete?
 2. Which bugs are safe to fix? (Some users of the software may be depending on the bug and treating it as a feature.)
 3. Which users need to be consulted before making changes to behavior?
- 
- A large yellow triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

1.4.2 Knowledge silos

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

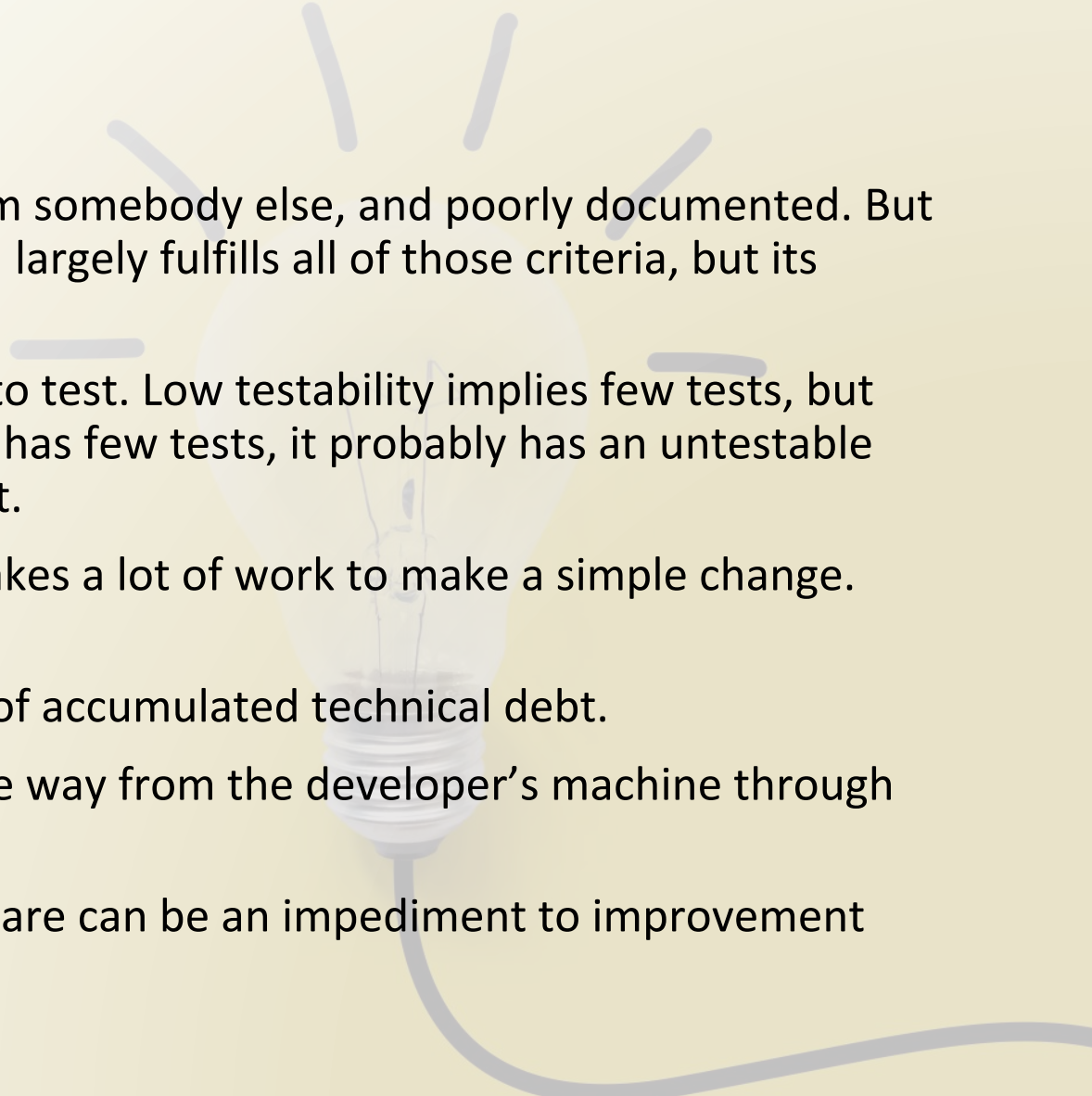
1. Domain information about the users' requirements and the functional specifications of the software
2. Project-specific technical information about the software's design, architecture, and internals
3. General technical knowledge such as efficient algorithms, advanced language features, handy coding tricks, and useful libraries

1.4.2 Knowledge silos

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

1. Lack of face-to-face communication
2. Code ego.
3. Busy-face

1.5 Summary

1. Legacy software is often large, old, inherited from somebody else, and poorly documented. But there are exceptions to the rule: the Linux kernel largely fulfills all of those criteria, but its quality is high.
 2. Legacy software often lacks tests and is difficult to test. Low testability implies few tests, but the converse is also true. If a codebase currently has few tests, it probably has an untestable design and writing new tests for it is thus difficult.
 3. Legacy code is often inflexible, meaning that it takes a lot of work to make a simple change. Refactoring can improve the situation.
 4. Legacy software is encumbered by years' worth of accumulated technical debt.
 5. The infrastructure on which the code runs, all the way from the developer's machine through to production, deserves attention.
 6. The culture of the team that maintains the software can be an impediment to improvement
- 



The End