Software rot

Software rot, also known as **bit rot**, **code rot**, **software erosion**, **software decay**, or software entropy is either a slow deterioration of software quality over time or its diminishing responsiveness that will eventually lead to software becoming faulty, unusable, or in need of upgrade. This is not a physical phenomenon: the software does not actually decay, but rather suffers from a lack of being responsive and updated with respect to the changing environment in which it resides.

The *Jargon File*, a compendium of hacker lore, defines "bit rot" as a jocular explanation for the degradation of a software program over time even if "nothing has changed"; the idea being this is almost as if the bits that make up the program were subject to radioactive decay.[1]

Contents

Causes

Several factors are responsible for software rot, including changes to the environment in which the software operates, degradation of compatibility between parts of the software itself, and the appearance of bugs in unused or rarely used code.

Environment change

When changes occur in the program's environment, particularly changes which the designer of the program did not anticipate, the software may no longer operate as originally intended. For example, many early computer game designers used the CPU clock speed as a timer in their games,[2] but newer CPU clocks were faster, so the gameplay speed increased accordingly, making the games less usable over time.

Onceability

There are changes in the environment not related to the program's designer, but its users. Initially, a user could bring the system into working order, and have it working flawlessly for a certain amount of time. But, when the system stops working correctly, or the users want to access the configuration controls, they cannot repeat that initial step because of the different context and the unavailable information (password lost, missing instructions, or simply a hard-to-manage user interface that was first configured by trial and error). Information Architect Jonas Söderström has named this concept *Onceability*,[3] and defines it as "the quality in a technical system that prevents a user from restoring the system, once it has failed".

Unused code

Infrequently used portions of code, such as document filters or interfaces designed to be used by other programs, may contain bugs that go unnoticed. With changes in user requirements and other external factors, this code may be executed later, thereby exposing the bugs and making the software appear less functional.

Rarely updated code

Normal maintenance of software and systems may also cause software rot. In particular, when a program contains multiple parts which function at arm's length from one another, failing to consider how changes to one part affect the others may introduce bugs.

In some cases, this may take the form of libraries that the software uses being changed in a way which adversely affects the software. If the old version of a library that previously worked with the software can no longer be used due to conflicts with other software or security flaws that were found in the old version, there may no longer be a viable version of a needed library for the program to use.

Classification

Software rot is usually classified as being either **dormant rot** or **active rot**.

Dormant rot

Software that is not currently being used gradually becomes unusable as the remainder of the application changes. Changes in user requirements and the software environment also contribute to the deterioration.

Active rot

Software that is being continuously modified may lose its integrity over time if proper mitigating processes are not consistently applied. However, much software requires continuous changes to meet new requirements and correct bugs, and re-engineering software each time a change is made is rarely practical. This creates what is essentially an evolution process for the program, causing it to depart from the original engineered design. As a consequence of this and a changing environment, assumptions made by the original designers may be invalidated, introducing bugs.

In practice, adding new features may be prioritized over updating documentation; without documentation, however, it is possible for specific knowledge pertaining to parts of the program to be lost. To some extent, this can be mitigated by following best current practices for coding conventions.

Active software rot slows once an application is near the end of its commercial life and further development ceases. Users often learn to work around any remaining software bugs, and the behaviour of the software becomes consistent as nothing is changing.

Examples

AI program example

Many seminal programs from the early days of AI research have suffered from irreparable software rot. For example, the original SHRDLU program (an early natural language understanding program) cannot be run on any modern day computer or computer simulator, as it was developed during the days when LISP and PLANNER were still in development stage, and thus uses non-standard macros and software libraries which do not exist anymore.

Online forum example

Suppose an administrator creates a forum using open source forum software, and then heavily modifies it by adding new features and options. This process requires extensive modifications to existing code and deviation from the original functionality of that software.

From here, there are several ways software rot can affect the system:

- The administrator can accidentally make changes which conflict with each other or the original software, causing the forum to behave unexpectedly or break down altogether. This leaves them in a very bad position: as they have deviated so greatly from the original code, technical support and assistance in reviving the forum will be difficult to obtain.
- A security hole may be discovered in the original forum source code, requiring a security patch. However, because the administrator has modified the code so extensively, the patch may not be directly applicable to their code, requiring the administrator to effectively rewrite the update.
- The administrator who made the modifications could vacate his position, leaving the new administrator with a convoluted and heavily modified forum that lacks full documentation. Without fully understanding the modifications, it is difficult for the new administrator to make changes without introducing conflicts and bugs. Furthermore, documentation of the original system may no longer be available, or worse yet, misleading due to subtle differences in functional requirements.

Refactoring

Refactoring is a means of addressing the problem of software rot. It is described as the process of rewriting existing code to improve its structure without affecting its external behaviour.[4] This includes removing dead code and rewriting sections that have been modified extensively and no longer work efficiently. Care must be taken not to change the software's external behaviour, as this could introduce incompatibilities and thereby itself contribute to software rot.

See also

- Code smell
- Dependency hell
- Software bloat

- Software brittleness
- Software entropy

References

1. Raymond, Eric. "Bit rot" (http://www.catb.org/jargon/html/B/bit-rot.html). *The Jargon File*. Retrieved 3 March 2013.
2. Inc, Ziff Davis (1992-01-28). *PC Mag* (https://books.google.com/books?id=uEkrL23rU98C&pg=PT303). Ziff Davis, Inc. p. 286.
3. Jonas Söderström. "Onceability: The consequence of technology rot" (http://inuseful.se/onceability-the-consequence-of-technology-rot).
4. Fowler, Martin (October 11, 2007). "What Is Refactoring" (http://c2.com/cgi/wiki?WhatIsRefactoring). Retrieved 2007-11-22.