Software Maintenance
Life after the ship has sailed

Overview

Software maintenance: modifying a system after ship

What are the categories of maintenance activity?

What ideas and techniques are important for effective software maintenance?

Categories of software maintenance

Corrective - fixing bugs

Perfective - satisfying new or changing use cases

Adaptive - rolling with the punches

Preventative - making it easier to do all of the above

Corrective maintenance

A typical bug report:

Repro case: "Apply a formatting change to an existing document, click save and then refresh the page."

Observed behavior: "The formatting changes are gone."

Expected behavior: "When I refresh the page after clicking 'save', all of my changes (formatting or otherwise) are present."

Typical corrective maintenance flow: reproduce the bug, write a test for it, fix the bug, deploy and verify the fix.

Perfective and Adaptive Maintenance

Perfective: adding new features, responding to changing requirements

Adaptive: responding to changing environment (new technologies, change in dependencies, performance degradation)

This stuff happens all the time.

Structuring code for maintainability

Two important ideas:

Single responsibility principle

Dependency inversion

Imagine an application that modifies objects and persists them using a database connection...
class WidgetMill:
    @classmethod
def update_capacity(cls, new_capacity):
        mill = session.query(WidgetMill).first()
        mill.capacity = new_capacity
        session.add(mill)
        session.commit()

if __name__ == '__main__':
    WidgetMill.update(10)

Violates the single responsibility principle
Two things are tightly coupled that might change for different reasons
Limits: adaptability of code, efficient use of resources...

Technical debt
Roughly: work that accrues by taking shortcuts that fall short of engineering best practices [Allman]

Often arises from tension between best practices and other constraints (ship date, cost, skill of engineers) e.g.:
Not using an abstraction because it's too hard to figure out
Skipping or scrimping on documentation (both internal and external)
Using a simple but inefficient algorithm to save time

Technical debt != Maintenance work
Some tasks that are rightfully considered ordinary maintenance are routinely miscategorized as tech debt
e.g. enlarging a database to handle more data
Worth speaking carefully about tech debt, otherwise we risk pathologizing a normal part of the software life cycle

Key manifestation: systems (code, workflows, dependencies) that resist change

Financial debt vs. technical debt

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>&lt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not necessarily bad</td>
<td>Not necessarily bad</td>
<td></td>
</tr>
<tr>
<td>Must be repaid</td>
<td>Often but not always repaid</td>
<td></td>
</tr>
<tr>
<td>Repaid with interest</td>
<td>Repaid with interest</td>
<td></td>
</tr>
<tr>
<td>You take it on, you pay it off</td>
<td>You take it on, you (often) hand it off</td>
<td></td>
</tr>
</tbody>
</table>
Example: UC Berkeley’s CalMail failure (2011)

Multiple, simultaneously RAID failures during usage spike

CIO: “I made the decision not to spend the million dollars to upgrade CalMail software for only 12 months of use given our plan to migrate to new technology. We were trying to be prudent given the budget situation, (but) in retrospect it would have been good to have invested in the storage upgrade so we would have avoided this crisis.”

Turned out to be a bad gamble, but (arguably) not irresponsible

Dimensions of tech debt [Fowler]

<table>
<thead>
<tr>
<th></th>
<th>Reckless</th>
<th>Prudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliberate</td>
<td>“We don’t need no stinkin’ tests!”</td>
<td>“We must ship now and deal with the fallout or we go out of business.”</td>
</tr>
<tr>
<td>Inadvertent</td>
<td>“What’s the single responsibility principle?”</td>
<td>“Now we know how we should have done it.”</td>
</tr>
</tbody>
</table>

The problem of other people’s code

Often very hard to understand context and tradeoffs being faced when debt was incurred

Code is harder to read than to write

Result: “this code sucks!”

Resist this line of reasoning.

We’ll just rewrite it from scratch!

Bad idea.

- All those warts and hairs that have sprouted all over the place? Those are bug fixes. They took weeks/months/years of real usage to discover.
- When you throw away code you throw away all accumulated knowledge and competitive advantage.
- No good reason to believe you’ll ultimately fare any better.

Much better idea to carefully replace parts of the system in piecemeal fashion.

Recap

Software maintenance: modifying a system after ship

Corrective, perfective, adaptive

Good software architecture matters! e.g. SRP, DI

Understanding and managing tech debt effectively will make you a much better engineer