Software Evolution

Software Architecture, Process, and Management

In this lecture we will look at some of the issues involved with the maintenance of software in the long-term. Much of what we have spoken about has been large-scale but a contributing factor in the difficulty of creating correct real-world software is that it must be maintained after its initial deployment. In this lecture we discuss how to handle changes to code over time, mostly for pre-existing (legacy) systems but also over the course of initial development.

Why Maintenance?

- Change is inevitable
- Even though software doesn’t wear out in the same way as physical artifacts, it still needs to be:
  - fixed for flaws that have been discovered,
  - adapted to changing needs,
  - improved in performance or maintainability,
  - improved by fixing bugs before they activate

Physical vs Intangible

- Physical projects such as buildings usually see maintenance as fixes for things which were once working but have now broken
- Software is, effectively, imperishable, so fixes are for things which were always broken
- Thought it may have become inappropriate because of the changing environment
- This, permanence cuts both ways, software is not as easily adapted by the users as some physical artifacts

Reading and Writing

- It is harder to *read* code than it is to *write* it
- Which contributes to the difficulty in re-using code
- It also makes the urge to re-invent new code very difficult to resist
- If it were just as easy to re-use code as to write new code, then we would not need principles such as *DRY*
- Unfortunately, it is generally easier to just write new code, than it is to refactor and reuse existing code
- Even more unfortunately, this then makes the result even more difficult to understand, since there are now two solutions which need to be understood

Existing Code

- Because it is so difficult to understand existing code, it is often seen as somehow worse than new code
- It is also easy to justify this to ourselves:
  - If we did not write the old code it is very tempting to believe that it is the result of a less adept designer
  - Old code contains remnants of bad design decisions which we now know how to avoid
  - Old code was designed to work in the world as it was then, the world has moved on, new technologies exist, old ones are unsupported
- **However**, old code has been used and tested

Even New Systems Change

- Even brand-new systems will often or typically need to:
  - Interface with existing components that cannot be replaced (usually undocumented, sometimes without source code)
  - Reproduce functionality of old, badly designed systems in current use, without breaking anything
  - Go through many revisions during development
- This is related to not throwing away existing *working* code
- Not just code but practices might seem inefficient but may be that way for some (likely undocumented) reason

Software Maintenance

- The ways in which we do this:
  - Reverse engineering Analysing an existing system, e.g. to create accurate documentation
  - Refactoring Cleaning up code without modifying functionality
  - Reengineering Creating a maintainable system out of an unmaintainable one, at a larger scale than refactoring
- Commonly, more resources, and specifically time, are spent on these activities than on new development

Why Maintenance is Difficult

- Maintenance is seen as uninteresting and low-status: hard to get the best developers to spend long on a maintenance team
- There is always more work than can be done, so corners tend to be cut
- Even if resource isn’t an issue, the intention behind the original design is easily lost
- So the software gradually loses its integrity: architectural degradation

Why does software spoil

- Why does software spoil in the first place?
- Recall the feature matrix; other software may be more appealing due to its advanced feature set
  - Even if it lacks basic features and/or is of lower quality
- Commercial software is often pressured into adding features otherwise customers wouldn’t upgrade
- Letts’s Law: “All programs evolve until they can send email.”
- Zawinski’s Law: “Every program attempts to expand until it can read mail.”

Why does software spoil

- Strangely though, open source software is by no means immune to this
- Firefox was originally a *"cut-down lightweight"* version of Mozilla
  - And yet the major complaints of Firefox continue to be related to bloat/memory hungry/slow
- Eclipse; what are you doing? STAHP!
  - 4.2 (Juno) is routinely bemoaned for poor performance
  - "It takes 30 seconds to open a file"
  - Many users reverted back to 3.8

Broken Window Theory

- A factory shuts down and ceases to operate
- It sits for months without any activity
- One day, some drunk or errant youth, breaks a window, perhaps accidentally
- Within a short time, all the windows are broken

Managing Change

- Put a process in place for software evolution:
  1. Bug report/change request submitted via issue tracking tool
  2. Show-stopping problems get dealt with immediately
  3. Other issues are classified and prioritised
  4. Subsequent software releases incorporate fixes for a selection of reported issues
- Trade-off: fixing old issues vs. introducing new functionality
- Note, that this trade-off is artificial, and only exists if one insists on distinguishing between bugs and feature requests
- Some *issues* are cheaper to live with
  - Some “bugs” are less urgent than some “feature requests”
  - Some “feature requests” are less urgent than some “bugs”

Legacy Systems

- A legacy system is one that is difficult to evolve
- Cynical view: legacy = any system actually being used:
  - If it isn’t being used it is simply thrown-away forgotten code
  - If it is being used, it is so *despite* being difficult to evolve
  - Systems are obsolete as soon as they are shipped - technology evolves, requirements change, etc., yet:
  - It is always difficult to make changes without disrupting existing users
- Without constant vigilance, continuous refactoring, and reengineering, this tradeoff results in a series of small, scattered patches that eventually destroy system integrity and make systems unmaintainable

Testing

- Another cynical definition: legacy = any system without a complete test suite
- Before you start, make sure you have a comprehensive test suite
  - If not, build one
- Without good tests, making changes is scary, so people become conservative rather than doing large-scale maintenance like refactoring

Testing

Before you start: Testing

- To remind you of the types of testing:
  - Unit testing: Conformance of module to specification
  - Integration testing: Checking that modules work together
  - System testing: System rather than component capabilities
  - Regression testing: Re-doing previous tests to confirm that changes haven't undermined functionality
    - Whenever a bug is fixed, a regression test should be added to test that that bug is not re-introduced later

Regression Testing

- Crucial for maintenance: Build up an automated library of tests that are run regularly to uncover newly introduced bugs
- For a legacy system, often one slowly adds unit tests to the regression test suite as one understands bits of the code
- Simple way to blindly generate regression tests: collect output from numerous runs of the current software, assuming whatever it does has been good enough so far, then only investigate when the output changes
- Source code control can even allow you to *retrospectively* add tests for things you didn’t think of

Refactoring

- Refactoring: improving the existing design without adding functionality
- Assume that we have either a partially implemented system or a legacy system to which we would like to add a feature
- Ideal: Just find where the new feature would go, and write the appropriate bit of code
- Actual: Extensive changes are often needed to the existing design before the new feature can be added
  - This is common because whatever the desired feature is, it possibly does not exist yet **because** of the existing design
- Refactoring helps avoid doing scattered patches, to keep the overall structure clean

Refactoring Approach

- Whenever the current design makes it unwieldy to implement a desired function or feature:
    1. Step back and re-design the existing code so that it will make the feature easy to add
    2. Make sure that the code meets the same tests as before, i.e., provides the same functionality
    3. Integrate the *refactoring* changes with the rest of your team
    4. Make the change, pass the tests, and integrate again

Refactoring and Testing

- Refactoring is much easier with good regression tests
- If none exist, first add tests for all the functionality you are planning to modify, and make sure that they are fully automated
- The tests can verify that a refactoring does not change the program behavior
- Typically: run tests as-is before and after refactoring, then modify the code and the tests, and again verify the tests run ok

Reengineering Legacy Code

- Assume that we wish to add features to a legacy system that has a complicated, suboptimal design
- Refactoring + testing approach:
  1. Set up tests to capture current behavior (time-consuming)
  2. Gradually refactor as code is understood (also slow)
  3. Once the design is relatively clean and appropriate for the types of changes now expected, start adding features (now easy)
- Benefit: it's usually obvious what to do next
- Disadvantage: A lot of time is spent whilst adding no new features and the customer is potentially waiting

A Complementary Technique

- Sometimes you want to save, but not continue to modify, a legacy system or component
- E.g., it’s written in an obsolete language, and/or it is incomprehensible (**but apparently correct!**)
- You can use the *Adapter* pattern
    - wrap it in a well-defined interface (using a foreign function interface if necessary) usable from a modern language
- All future code interacts with the legacy only through the adapter
- Limited to cases where you can isolate the valuable legacy code

Adapter Pattern

- Draw a fence around legacy code, and not attempt to improve it
- Use for an obscure, undocumented but reliable component for which refactoring would be difficult
- Wrap the old FORTRAN or COBOL component as a nice-looking object in the new development language, use the adapter in the future, and never touch the old code again
- Benefit: Avoids the broken window syndrome: Prevents the old bad code from tainting new good code
- Benefit: Someone could independently work on replacing the legacy component

Adapter Pattern

- It may be possible to run the legacy code as a separate process
- A common approach is to run the legacy code as a web-service exporting either a web-interface or web-API
- This means that the legacy code needs only to be installed on the web-service, and not, for example, on every client machine
- Can be used to make a legacy program available on, for example, a tablet

Tools for Legacy Systems

- Program comprehension tools help explore an unfamiliar program
  - Many techniques can be used, providing different information useful in different circumstances
  - For example, a slicing tool can be used to identify which lines of a program affect the value of a specific variable at a point of interest

Tools for Legacy Systems

- Reverse engineering tools construct a high level view of a system from a low level one
  - This may be source code from object code (useful if source code has been lost),
  - a call graph in which nodes representing modules, or
  - functions are connected if one calls the other, or
  - perhaps a UML class diagram

Keeping the System Running

- New projects are often designed to replace an existing body of lousy code in current use, but adding features
- Many such projects fail by being overly ambitious
- Often, the developers go far out on a limb adding fun new features and improvements, before the system has been put into real use
- The longer this goes on, the lower the likelihood of the new system ever being used
- The alternative of making small changes that keep the old system working continuously is very hard, but it's a good way to minimise the chance of total failure

One Component at a Time

- Be like Trigger and replace only the part of the legacy system that you need to, in order to perform the currently required maintenance
- Over time, you may replace the entire system, and in particular then you may have ported the entire system to a new language
- However, with each component replacement the risk attached is limited to the functionality connected with that component
- This kind of modular/incremental complete re-write is of course made easier if the legacy code is modular in the first place
- One possibility is to refactor the legacy code, before replacing it one component at a time
- But, importantly, you are not scrapping the entire code in one go

Technical Debt

- In software development we often face a similar choice
- There is some functionality, or bug-fix that we wish to do
- Doing so properly, requires a re-design of some of your code:
  - This may take a long time
  - There may be a quicker solution, but it will leave the code base, in a less desirable state
- When you do this, your code-base now has some *Technical Debt*:
  - There is some amount of time/effort that you later need to pay back in order to recover the integrity of your design

Technical Debt

- This might be worth it, for example in the case that the fix is a security fix and you want a patch for it as quickly as possible
- However, the metaphor extends to *interest*
- If you do not pay back your technical debt immediately, subsequent modifications to your code may take more effort, than they would have, because your code base is not in a desirable condition
- This extra effort, is the *technical interest* you are paying on your *technical debt*

Compound Interest

- In both financial and technical debt if you do not take steps to reduce your debt, then the interest accumulates and is known as *compound interest*
- This means you are paying *interest on your interest*
- This leads to exponential growth
- Humans have a hard time understanding exponential growth
- As the chessboard legend demonstrates

Bankruptcy

- When people struggle to pay back the loans they currently have, they may choose to take out additional loans in order to pay the current ones
- Often these additional loans will be more expensive, that is the interest rate will be higher
- If you do this enough your credit rating will become so poor that no one will be willing to lend any amount of money at any interest rate to you
- The amount you are earning, is less than the *interest* you have to pay on your loans, so you have no hope of re-paying your loans

Technical Bankruptcy

- Similarly, if you continue to make changes to your source code without repaying any of the technical debt, changes will become more difficult to make
- Even making the *quick* fixes takes a long time
- Eventually the state of your source code is so bad, that making any changes, would require as much time as re-starting the whole development from scratch
- At which point you have hit technical bankruptcy

Technical Debt

- Of course this is just a *metaphor*
- In particular, it is difficult to judge just exactly how much technical debt you have, and whether or not you have reached the point of technical bankruptcy
- You will rarely know what the absolute correct design is, or be able to estimate how long it would take you to achieve that
- Or indeed estimate how long updates would take in either the case of your current technical debt, or lower levels
- But note; if we cannot reliably calculate compound interest using concrete values, what chance do we have using vague notions, approximations and estimates?

Lehman’s Laws

- Manny Lehman, the “Father of Software Evolution”, wrote many papers from the mid 70s onward proposing rough “Laws of Software Evolution”
- Not always clear whether these are based on data and it is likely that *“laws”*, overstates the case
- These apply to a real-world system actually evolving while being used, not a static, formally specified system
  - A distinction he called S-type and E-type systems

Lehman's Laws

S-type and E-type Systems

- S-type systems are those that are, or can be, formally specified
  - Moreover an S-type system implementation can be tested against that formal specification for correctness
- E-type systems are those being used and adapted in real-world scenarios
  - It generally does not make sense to talk of the *correctness* of an E-type system, but rather its *suitability*
- It is possible to construct E-type systems using S-type systems as formally specified components
- The following laws apply to E-type systems

Lehman’s Laws 1/8

1. Continuing Change
    - System must be continually adapted else it becomes progressively less satisfactory in use
    - The Y2K bug being a rather sharp-edged example of this phenomenon

Lehman's Laws 2/8

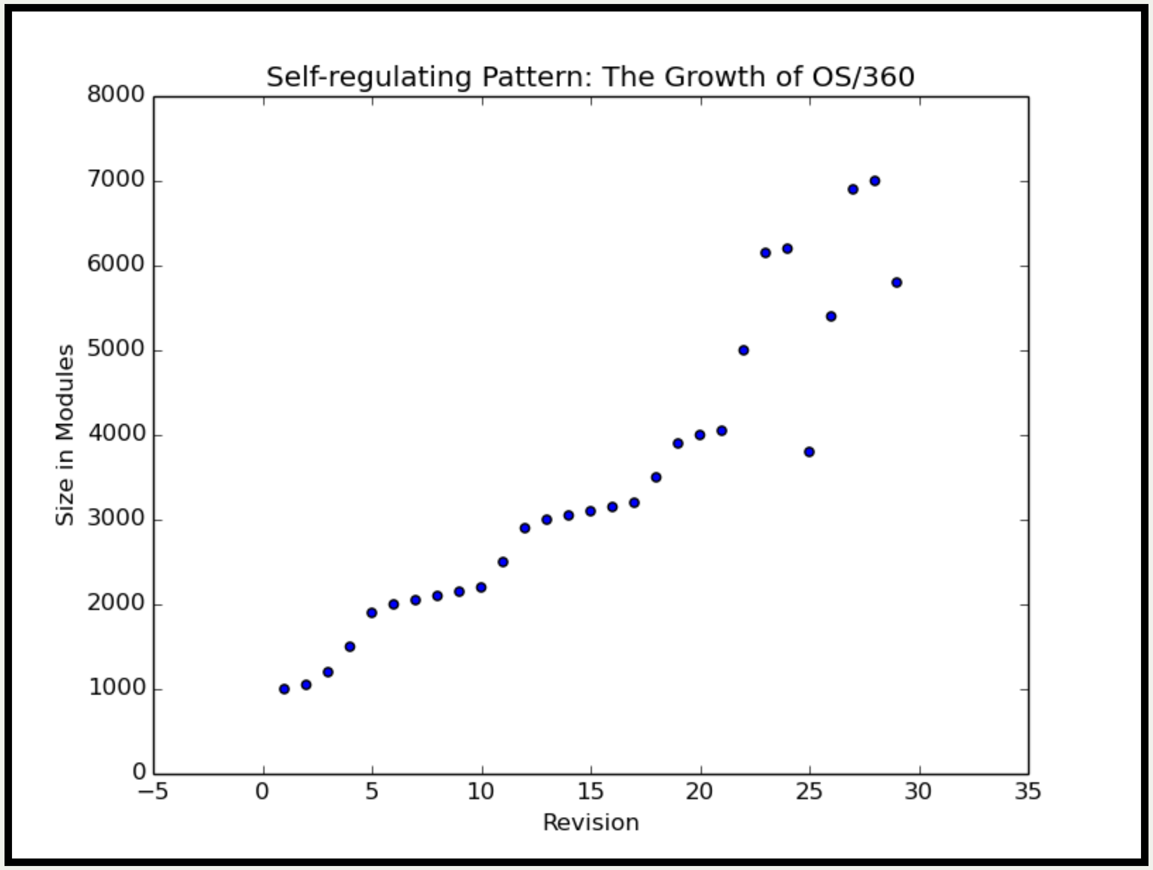
2. Increasing Complexity
    - As it is evolved its complexity increases unless work is done to reduce it
    - Design integrity is **not** inductive
    - The sum of a series of *good* changes does not necessarily equate to a good overall change
    - Can you do 365 press-ups on December 31st?

Positive + Negative Feedback

- Feedback can also occur in the positive direction, more output gives rise to a further *increase* in output
- Systems which display both positive and negative feedback may be self-regulating
- Some property, for example *body temperature* although constantly changing, has a long-run average
- Such systems are said to be in homeostasis
- The property which remains constant may be a **rate** of change

Lehman's Laws 3/8

3. Self Regulation
   - Global Evolution processes are self-regulating



Lehman’s Laws 4/8

4. Conservation of Organisational Stability
    - Average activity rate in its process tends to remain constant over system lifetime or segments of that lifetime
    - This is observational activity
    - It could be that it is becoming more expensive to keep up that activity rate, but that that expense is covered

Lehman's Laws 5/8

5. Conservation of Familiarity
   - <u>In general, the average incremental growth (growth rate trend) tends to decline</u>
   - Developers, sales, personnel, users etc. must maintain mastery over the software product
   - Excessive growth tends to diminish that mastery
   - Hence the average incremental growth rate remains invariant as the system evolves

Lehman’s Laws 6/8

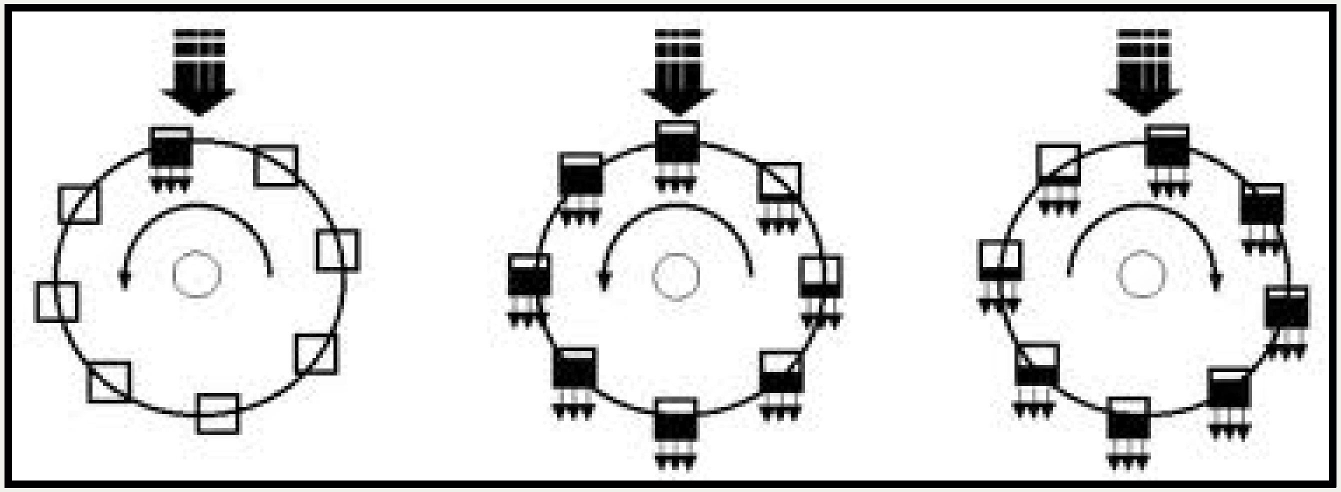
6. Continuing Growth
    - <u>The functional capability must be continually enhanced to maintain user satisfaction over system lifetime</u>
    - Almost a restatement or specialisation of law 1
    - Essentially users demand improved functionality over time, rather than simply improved quality of existing functionality

Lehman’s Laws 7/8

7. Declining Quality
   - Unless rigorously adapted to take into account changes in the operational environment, the quality will appear to be declining as it is evolved
   - Again something of a re-statement/specialisation of law 1

Lehman's Laws 8/8

8. Feedback System
   - Evolution processes are multi-level, multi-loop, multi-agent feedback systems
   - Something of a summary law of all the previous laws
   - Essentially we have a *"chaotic"* system
     - *"Chaotic"* in the strictly technical sense related to a dynamic system



Maintenance

- Maintenance is essentially development plus understanding the existing product
- Maintenance is a solution not a problem
- Better software engineering leads to *more* maintenance *not* less
  - This may be counter-intuitive, however, when maintenance is seen as a solution rather than a problem the more of it we do the more successful our product is

Evolution and Agile Methodologies

- Maintenance takes up a large proportion of the overall development time
- Agile methodologies could be seen as starting with the maintenance
- In this way you do not need to separate out developing new code from maintaining old code, it’s all the same work, improving on the existing code, regardless of how old that code is