Software Architecture, Process and Management
Software Evolution

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/sapm
Semester Two 2012-13
Quick announcement:

Level 10s, don’t forget final deadline for most of your second blog posts is Monday 18th March 4pm.

You can comment on blog posts for two weeks after that, until Monday 1st of April 4pm.
public String readAuth() throws IOException {
    int s;
    ByteArrayOutputStream bao = new ByteArrayOutputStream();
    while ((s = is.read()) != -1) {
        bao.write(s);
        if (!is.ready())
            break;
    }
    if (s == -1) {
        return null;
    }
    String result = bao.toString();
    return result;
}
SAPM: What does this code do?

It turns a character array into a string whilst also deleting any empty elements, or in code:

```java
String bufferString = new String(buffer).trim().replaceAll(" ", "");
```
It’s 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
Old code is often seen has somehow worse than new code
  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
  Old code is bloated, buggy and inefficient

However, old code has been used and tested
SAPM: Sudoku

Difficulty Level ★★★★☆
There isn’t a huge amount of reason to assume that you will do better *this* time
If you throw away your old code you are throwing away a lot of work

- bug fixes
- undocumented design decisions
- human factors difficult to capture in requirements and in any case are likely undocumented, eg. *Delete* button moved away from *Save* button.

The alternative is to perform software maintenance on the existing code

In your university courseworks, even the system design project, you have almost zero experience of this, since your code, once submitted is never required to be updated

In the system design project (level 10s) you all had the opportunity to re-use existing code, and it’s my impression that the actual amount of re-used code could be very well approximated with the number zero
What we wish to do is maintenance:
- Fixing bugs and adding new features
  - Which to me are one and the same thing but others disagree
- Adapting when external components change
- Adapting to external conditions such as new platforms

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
SAPM Maintenance

- Maintenance in software is awesome
- Software’s unique solution to the problem: “We built this thing but now we wish we had built something a little different”
- Remodelling a tangible product is difficult; a software product can be modified **comparitively** easily
SAPM: Product Recalls

- 2009-2011 Toyota Vehicle Recall
- Three separate recalls of faulty (and dangerous) vehicles at great expense to the manufacturer and inconvenience to the customers

  “A new car built by my company leaves somewhere traveling at 60 mph. The rear differential locks up. The car crashes and burns with everyone trapped inside. Now, should we initiate a recall? Take the number of vehicles in the field, $A$, multiply by the probable rate of failure, $B$, multiply by the average out-of-court settlement, $C$. $A \times B \times C$ equals $X$. If $X$ is less than the cost of a recall, we don’t do one.” — Narrator, Fight Club

- In contrast software which is faulty can be patched
SAPM: 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,
  - corrective maintenance
  - adapted to changing needs,
  - adaptive maintenance
  - improved in performance or maintainability,
  - perfective maintenance
  - improved by fixing bugs before they activate
  - preventative maintenance
- The refactoring, reverse engineering, re-engineering all support these operations
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
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
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 (maybe even by a binary patch);
  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”
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.”
Paint Shop Pro releases

<table>
<thead>
<tr>
<th>Version</th>
<th>Year</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>1993</td>
<td>0.4 MB</td>
</tr>
<tr>
<td>3.11</td>
<td>1995</td>
<td>1.8 MB</td>
</tr>
<tr>
<td>4.12</td>
<td>1997</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>5.0</td>
<td>1998</td>
<td>6.7 MB</td>
</tr>
<tr>
<td>6.0</td>
<td>1999</td>
<td>≈14.2 MB</td>
</tr>
<tr>
<td>7.0</td>
<td>2000</td>
<td>32 MB</td>
</tr>
<tr>
<td>8.0</td>
<td>2003</td>
<td>54 MB</td>
</tr>
<tr>
<td>9.0</td>
<td>2004</td>
<td>108 MB</td>
</tr>
<tr>
<td>10.0</td>
<td>2005</td>
<td>106 MB</td>
</tr>
<tr>
<td>11.0</td>
<td>2006</td>
<td>211 MB</td>
</tr>
<tr>
<td>12.0</td>
<td>2007</td>
<td>326 MB</td>
</tr>
</tbody>
</table>

Data from www.oldapps.com
Welcome to StarOffice!

Every time you start StarOffice, the Help Agent will display some tips. These tips contain useful information that will ease your work in StarOffice.

If you do not want the tips to be displayed, check the Don't display tips field and you won’t see them when the program is next started. Use the Close button to close the Tip window. You can control the Tip window view under Tools - Options... - General.
Strangely though, open source software is by no means immune to this

Firefox was originally a cut-down lightweight version of mozilla

- And yet among those who have migrated from Firefox to Chrome have done so complaining of it being: bloated/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 have reverted to 3.8
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
SAPM: Solution Landscape
Before you start: Testing

▶ Another cynical definition: legacy = any system without a complete test suite
▶ Without good tests, making changes is scary, so people become conservative rather than doing large-scale maintenance like refactoring
▶ 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 add tests for things you didn’t think of retrospectively
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 doesn’t 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 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.
Revision Control and Refactoring

- Adding a feature using git, testing, and refactoring:
  1. `git commit -a` (Commit all outstanding edits)
  2. emacs/vim/IDE (Refactor, not changing behavior at all)
  3. `make tests` (Regression test)
  4. `git status` (refactoring changes, large)
  5. `git commit -a -m "No visible changes"`.
  6. emacs/vim/IDE (Add new feature)
  7. `make tests` (Regression test)
  8. `git status` (added feature changes, small)
  9. `git commit -a -m "Added feature Y"`.
 10. emacs/vim/IDE (add test(s) for new feature)
 11. `make tests` (Regression test, new test(s) included)
 12. `git commit -a -m "Added tests for feature Y"`

- Goal: validate nearly all changes against existing tests, then debug new user-visible change/feature by itself
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
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 design 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.
SAPM: 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
- Avoids the broken window syndrome
- Benefit: Keeps old bad code from tainting new good code
- Benefit: Someone could independently work on replacing the legacy component
SAPM: 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, every client machine.
- Can be used to make a legacy program available on, say, a tablet.
SAPM: Adapter Pattern
SAPM: Adapter Pattern
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.

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.
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.
If you think you need an entire re-write, consider being like Trigger:
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.
Maintenance typically consumes 40 to 80 per cent of software costs

Software maintenance is largely about enhancement rather than fixing old capabilities

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, when maintenance is seen as a solution rather than a problem the more of it we do the more successful our product is
Maintenance takes up a large proportion of the overall development time

Agile methodologies could be seen as starting with the maintenance

```python
1 print "Hello, World!"
```

That’s your project; first issue: It doesn’t do X, where X is the most important feature

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
SAPM: Maintenance is Hard
SAPM: Maintenance is Hard

- Code is the enemy, keep your code as small as possible
- Consider retiring your software
  - Google this week announced the retirement of Google Reader
  - Which has admittedly upset a number of users
Summary

- Change is inevitable — prepare for it
- Refactoring helps keep you (and your design!) sane when making changes
- Refactoring, testing, and revision control together make it much more practical to reengineer legacy code
- Iterative development/refactoring allows the project to avoid the cost of maintenance from spiralling out of control
- Code is your enemy, feature bloat is when you have too much code
- Users are impatient and want visible changes now, you must fight against this
  - “I want a gold plated toilet but it’s just not on the cards” — Austin Powers
SAPM — Related Reading

► Required Reading
  ► None

► Suggested Reading
  ► Refactoring: Improving the design of existing code, Fowler et al, 1999
  ► www.refactoring.com
Quick announcement:

Level 10s, don’t forget final deadline for most of your second blog posts is Monday 18th March 4pm

You can comment on blog posts for two weeks after that, until Monday 1st of April 4pm
Any Questions?