

Principles of Simple Software Development

Introduction

The following principles were developed in the course of the e-Administration of Teaching project^{1,2} at the University of Cambridge, funded by the JISC's Institutional Innovation programme³.

The project involved developing and deploying a piece of administrative software called the Teaching Office Database (TODB) to six departments. Each department required unique customisations in the software, but accepted all subsequent responsibility for maintaining and supporting it.

The concept of simple software was originally developed by Prof. Richard Prager at the Engineering Department⁴ and subsequently applied and refined by members of the project team, at the Centre for Applied Research in Educational Technologies⁵.

These principles should be used to guide development of software which will be deployed and maintained locally in university departments by non-professional programmers such as computer officers, competent academics, even summer students.

Principles

Appropriate to the task

Software involving workflow through numerous or systems or users, especially complex ones or those outside the maintainer's sphere of influence, is inherently unlikely to be 'simple'. Similarly for software which must be used by many different users, or which involves high-stakes or confidential operations.

No external libraries

Libraries are 'black boxes' for those unfamiliar with them and make it more difficult to build a mental model of the software. Application frameworks are bad for the same reason - they might be great for professional programmers, but they introduce a huge amount of noise for novices. External libraries get updated, maintained and retired, adding work for the diligent application maintainer, especially if they are popular targets for security hacks.

This rule is flexible, within reason - sometimes a library or API might be the only reasonable way of achieving something, Shibboleth login for instance, but the impact on the maintainer but always weigh heavy in the balance.

No abstractions

Abstractions are usually introduced to promote code re-use and support configuration for different users. Even if well explained and documented, they make it harder for someone unfamiliar with a program to understand how it works. Lots of 'big' projects are actually small, once all the abstraction and configuration capability needed to handle different user requirements is stripped out.

¹ <http://www.jisc.ac.uk/whatwedo/programmes/institutionalinnovation/modulareadminofteaching.aspx>

² <http://modular-e-admin.blogspot.com/>

³ <http://www.jisc.ac.uk/whatwedo/programmes/institutionalinnovation.aspx>

⁴ <http://www.eng.cam.ac.uk/>

⁵ <http://www.caret.cam.ac.uk/>

Novice-friendly

Professional software developers will want to apply their ideas about best practice, appropriate tools, and software architecture. All this draws on considerable background knowledge and context, without which none of these things makes much sense. Put best practices to one side and consider the novice user. How would someone whose background consists of “PHP in 24 hours” approach the problem?

The code is the configuration

Don't try too hard to bring all possible configuration parameters into one place. Parameters such as text labels clearly benefit from extraction, but creating configuration abilities beyond that basic level requires increasing abstraction and alternative execution paths in the code, placing a larger barrier in the path of subsequent maintainers and developers, and risking the counterproductive situation where understanding the configuration language is as difficult as understanding source code.

Users know what they're doing

I.E., users have a reasonable mental model of the process and data structures. The tool should support that mental model, and not attempt to second-guess or restrict their actions.

For example instead of enforcing strict referential integrity, TODB provides operations that assist the user in locating anomalies, e.g. a query that lists all jobs allocated to people who are not found in the database⁶. In a typical database scenario it would not be possible to allocate a job to a non-existent person, but this is possible on paper, and in the mind of the teaching administrator, so that to impose such a restriction in the software would only interfere with their working and thinking process.

⁶ A further example from TODB is that joins that might, in a typical system, be based on references and resource identifiers, are instead in some cases based on 'fuzzier' text-matches (based on regular expressions, or some text from side of the join appearing in a string in the other side of the join) that allow, for example, a relation to change from one-to-one to many-to-one without having to change the architecture of the software, nor the schema of the database, but only one or two queries in the PHP code. Very few domain rules apply – generally, users can enter anything into almost any field, in the same way that it is possible to write a date in a number column on paper. The software and queries are designed to be very robust with regard to data entry, and generally ignore data that do not make sense, rather than throwing errors. As the database is fundamentally simple (containing three main tables (people, jobs, units) which correspond almost directly to pages on the TODB system (view people, view jobs, view units), it is not difficult for users to understand how the database works. Understanding the database, and allowing flexibility on data entry, gives users the ability to find their own use model for the software to suit the processes of teaching allocation peculiar to their department, rather than forcing a particular way of working.