

The Trunkless Development Model

Contents

Introduction	1
Overview	2
Model summary.....	2
Prerequisites	2
Suitable projects	2
Model benefits.....	3
Model limitations	3
Caveats	3
Comparison of software development approaches	3
Fitness and distrust of centralised systems in the Cambridge University context	3
Quick and dependency-free operation.....	4
Iterative participation	4
Use of an initial prototype	4
Resource allocation considerations	5
Software simplicity and functionality	5
Software support.....	6
Project scale	6
Specification-variability considerations	7
Inter-instance compatibility.....	7

Introduction

The ‘trunkless’ model was developed and used 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, locally-maintained software customised from a common base was originally developed by Prof. Richard Prager at the Engineering Department⁴ in order to solve the problem of sharing the TODB with other departments, sharing a common problem but each having particular requirements, without requiring support from central IT services. The idea was subsequently applied

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/>

and refined by members of the project team, at the Centre for Applied Research in Educational Technologies⁵.

Readers may also be interested in the final report of the project, to which this white paper is a companion, along with a second document "Principles of Simple Software Development".

Overview

A method of spreading user-originated software in HEIs, which allows each user a 'made to measure' customised version of the software, for which they are subsequently able to take complete responsibility.

Model summary

1. Prototype developed by originating user
2. Other users interested
3. Central, one-off funding obtained for development
4. Software is genericised, with a view to making expected range of stakeholder-specific customisations manageable
5. Software deployed and customised user by user. Users and developers agree priorities for development.
6. Lessons are incorporated back in to generic version and pushed to previously-addressed departments
7. Detailed, tested documentation and handover to users

Prerequisites

- A working prototype exists
- At least some partner institutions have expressed an interest in evaluation with a view to adoption
- Institutional IT systems are unlikely or unable to assume similar functions in the project timeframe

Suitable projects

- Small to medium-sized applications in which the use of frameworks, large or numerous libraries, or substantial amounts of abstraction is unnecessary. It is worth noting that lots of 'big' projects are actually small, once all the abstraction and configuration capability needed to handle different user requirements is stripped out.
- Projects with a user base of approximately 3-20 users, each with distinct needs. If the numbers are larger or the requirements uniform then a traditional open-source approach is likely more efficient.
- The risk of having to apply similar and significant changes to all systems subsequent to deployment is small.
- It is not critical that all systems interoperate consistently. If it is, then centralised systems are likely to be more effective, as the overhead of 'softly' maintaining interfaces will offset the advantages of locally-maintained, locally-modified software.
- Administrative tools are in general likely candidates.

⁵ <http://www.caret.cam.ac.uk/>

Model benefits

- Software which, being created by academics, addresses their particular needs very well, can be genericised and then customised repeatedly for other users. Traditional, centrally budgeted and supported systems would not be able to provide such customisation at a reasonable cost owing to ongoing support requirements, and in any case are unlikely to invest a great deal of effort on behalf of relatively limited numbers of users, unless there are substantial financial or educational benefits.

Model limitations

- Originators are by definition able to resource a modest level of formal or informal development, and are likely to be able to do so again, if necessary. However, without the benefit of JISC or other capital funding it is not possible to re-engineer and distribute systems in this way.
- Less well-resourced adopters will be able to maintain the software, but not to extend or make significant modifications.
- The model is good as long as it is not subsequently necessary to apply similar but significant changes to all the deployed, customised systems, in which case support cost has been multiplied.
- The model accommodates large variations between sets of users, and becomes leaner, more efficient, and more like a conventional 'one size fits all' development approach as this variability decreases. It might become apparent that actually a one size fits all approach would have been appropriate, in which case architectural inefficiency might result from having followed the TODB approach initially.
- This kind of project might benefit from having fewer staff, for a longer period; or from being able to draw on a developer 'pool'. This would overcome difficulties in hiring great staff for a short term. Longer, lower intensity projects also allow more time to talk and work with other institutions, once the local systems and processes were established.

Caveats

Because project partner institutions are only now, at the end of the project, falling back on their own capabilities for maintenance and support, and because none save Engineering have as yet used the software over a complete academic year, and because developments in TIS and the student information system are still playing out, we can not say that evaluation is finished. It needs a long view over coming years.

Comparison of software development approaches

Fitness and distrust of centralised systems in the Cambridge University context

The University of Cambridge is, to a greater degree than most, a loose collection of independent or largely-independent colleges and departments. Each department (and each college, if the TODB is extended this widely) prides itself on its independence, so there is potentially large diversity in the way that teaching is organised in different departments. This independence is fiercely guarded as part and parcel of academic freedom. These conditions create considerable scepticism about centralised 'one-size-fits-all' software. The University's cultural memory records the deployment of the two largest and most centralised such systems - the finance and student information systems - as being painful experiences. This strengthens the distrust of centrally-provided software. The organic, per-department approach followed with the TODB avoids running foul of this by customising the software for each department and allowing each instance to be locally managed. From departments' point of view, this avoids the risk and overhead of engaging with central bodies who have many other priorities to balance and may deliver an unsatisfactory one-size fits all solution.

Quick and dependency-free operation

One of the advantages offered by small, independent and non-centralised software is that the usual dependencies that might apply bluntly in a large institution will not necessarily be enforced. For example, under most teaching circumstances, a member of staff will be appointed formally, will appear on a central staff database, and their information eventually trickles down to where it is needed. In a situation where a representative of industry gives a single lecture, they need not be formally appointed as a member of staff for the duration of that lecture. In the TODB, if the people list was 'formal' and enforced, it would be difficult to accommodate such people unless special provision was made in the system. By severing links with central systems, it is very easy to add that person as a member of staff – or even assign a job to them without them appearing in the people list. The work is accounted for. Centralised systems require far more prescription, consistency and formality in their operation in order to adequately handle their greater scope – and this is not necessarily how a Cambridge University department operates, or is willing to operate. The cost of course is the inability to synchronise information when this is desired – when a new member of staff joins, or someone retires.

Iterative participation

Rather than trying to develop a set of requirements from discussions about abstract concepts in the first phase of the project and then going ahead and developing the software, the constant to-and-fro between the developers and the project participants made for a very much richer interaction and engagement. It is felt that better software resulted – and not only this; as the stakeholders have been present in the software development process for up a to year already, those individuals probably feel that their contributions to the software have been substantial and worthwhile, and will feel a much greater desire to use the software.

The centralised, phased alternative is far less personal: each individual's contributions become diluted by time and the necessity to manage conflicting requirements for the software (where multiple stakeholders are consulted for the development of a single centralised piece of software). With the TODB approach, rather than an individual's desires having been 'taken into account', they are, in general, reflected in the implementation. That said, the phased/centralised model usually has one or more feedback iterations anyway; nevertheless, it is felt that this software development approach supported true engagement with the project's stakeholders. In fact, this could be considered an outstandingly successful aspect of the approach taken.

Use of an initial prototype

A number of participants expressed the view that an important success factor in this project was the availability from the start of a prototype system: the original, developed within Engineering. The prototype gave concrete form to what would otherwise have been a hard-to-visualise and far-off concept. It also meant that rather than being expected to develop *ab initio* for each department, customisation was limited to adapting the prototype, reducing to manageable proportions inter-departmental differences. It should be noted that the development and distribution of a 'prototype' originating in a particular department is a key part of this software development model. Developing a prototype remains a fairly substantial software development exercise, but where a prototype is available, this model is highly appropriate.

We have also seen, in the case of TIS, that software originated by individual academics or departments can be both motivation and specification for others, including central institutional IT departments, who wish to take on, extend, productionise and institutionalise the software.

Resource allocation considerations

It is worth noting that the trunkless model is not free - it simply offers an alternative division of resources. The cost is borne by the funder of initial customisation and roll-out - JISC in this case - and by departments, in slight but ongoing hosting and maintenance duties for their computer officers.

The TODB process consumed substantial resource in terms of project staff time. In future it should be considered whether beneficiary departments will realise sufficient value from the software to justify funding a developer for some time - two months in the case of TODB - to customise and support their adoption of it.

Without knowing the extent of stakeholder differences, it is difficult to predict how much software development work will be required for a 'trunkless' project. Unexpectedly high variability would require more customisation. Much of this risk is easily controlled by engaging potential adopter departments with a prototype before beginning the project, and setting expectations in terms of development priorities rather than an 'all you can eat' menu.

Even though structural differences may be present at department level, at a broader level University departments are roughly similar: students are lectured and examined by representatives of the department and colleges. Given this large commonality, it would seem likely that software that assisted and addressed teaching issues at this level would be useful. Generally speaking, larger budgets are allocated to projects with larger numbers of clients – so where a centralised systems is viable, it might be expected to be a more polished, feature-rich product.

Software simplicity and functionality

The 'locally-maintained' or 'trunkless' aspect of this unconventional software development approach requires a careful approach to software complexity. Given the profile of the expected person in charge of further development of TODB in a department – an interested academic, a computer officer, possibly a summer student – the software really does need to be simple and straightforward, placing a higher value on this than even 'best practice' computer science conventions (highly structured abstractions, data models, etc).

This leads to two issues for consideration:

- What is simple software? What are the conventions to be avoided? Essentially, how is the line drawn between acceptably-simple and unacceptably-complex coding approaches? In this project, the TODB had been initially developed by a non-expert, and was extended in a similar style. Although PHP and MySQL had already been chosen as the underlying technology for the TODB, these can be considered relatively easy technologies in which to acquire basic competence.
- To some extent, a tradeoff exists between complexity and functionality. This certainly does not hold in all cases, but there are definite situations where the code required to achieve a certain level of functionality will be more complex than might ideally be found in a 'simple, straightforward' system designed to be maintained and extended by non-experts. In practice, in such situations, either the complex functionality was not included in the software, or small sections of code within the TODB have complexity that exceeds the ideal target. Overall, however, it is felt that the software abides by the 'simple, straightforward' principle.

Further, computer scientists will argue that many of the conventions that have evolved in professional software development are intended to ease and simplify the software development process. Use of mature and proven software libraries comes at the cost of learning the libraries and applying some forethought to avoid situations where the use of the library limits possibilities; in return, it provides a clearer and more maintainable software source, which is also more likely to work properly because a

smaller proportion of the total product is new and relatively untested. Writing 'simple, straightforward' code – that avoids using libraries because of their learning curves – is actually more difficult, because developers are required to develop and test more from scratch. It is possible then that this approach is more costly in time and effort. Code that uses well-known libraries tends to be clearer and easier to follow if the reader is familiar with the libraries; in such cases, understanding the code of a completely built-from-scratch system can be a formidable task in itself.

In the specific case of TODB, it was noticed that in order to ensure that the database is as simple and straightforward as possible, the SQL queries in the code are more complicated than they might otherwise be. This appears to be another tradeoff, where the complexity is defined more by the functional requirements than by how the solution is implemented.

Readers may be interested in our blog post listing some principles of 'simple' software development⁶.

Software support

The development of a software product usually also develops a software support problem. For an institution like CARET, apart from allocating the resources required to liaise with users and write the code, people need to be trained for support as well. Criticism of the software development methodology used with the TODB has been leveled at the difficulty in supporting software that, at the time of release, has eight slightly different versions, and may have been independently modified further without knowledge of changes being fed back to the support team. The simple answer is – there is no support burden! The whole point of the trunkless, decentralised model is that departments are self-supporting. This is a 'launch and forget' model. Were such a model used more widely, it would be wonderful for software developers, who traditionally enjoy developing new things and despise the 'housework' of ongoing support. The temptation to compromise the constant focus on local maintainability must be held in check, however.

In practice, it is possible that departments who make few changes to the software might require some support. CARET will provide support as required to Cambridge departments using versions of the software, as part of our institutional contribution to the project. Those departments which engage with the software to the point of making substantial changes to the code are likely not to need support – as their own competence at that stage will be quite sufficient to address any TODB-specific support challenges they encounter.

The Google Code site⁷ used to make the generic code widely available also supports community features including a mailing list, wiki, and bug tracking, creating an infrastructure for community-based support in the style of a 'trunked' OSS project. Although in future it might be expected that some users will develop their TODB instances away from being in a position to offer mutual support, it might also be expected that they would be replenished by new adopters. Support within Cambridge will be mediated by a group mailing list and a backed-up CARET SVN repository containing all versions of the software, plus Cambridge-specific documentation.

Departments have been warned that although CARET will provide support this will not include further code customisations and will not be of the intensive level available during the project.

Project scale

The approach of developing a rough prototype and then working individually with eight departments approximately in parallel was excellent for a product of the relatively small scale and complexity of TODB.

6

7 <http://code.google.com/p/todb/>

Each instance of the TODB is somewhat different. Maintaining eight independent branches means each time a change is made to one that might be useful to all, a rather laborious merging process is required to ensure that the new functionality is available in other departments but does not overwrite some other department-specific changes. This was manageable for the e-Admin TODB project development, but if the application had been much larger, or if there had been many more departments involved, it might not have been. On this basis the approach is very favourable for small projects and may be untenable for larger ones.

The situation might be retrieved using the approach, previously discussed, of separately modularising 'core' and 'custom' code - as long as this can be successfully reconciled with the need to eliminate avoidable abstraction. It is worth noting also that Biochemistry, the last department to join the project, benefitted from a relatively mature TODB and required very few customisations. This suggests that the support requirement may not scale linearly with the number of departments, as early lessons are incorporated and it becomes progressively harder to find new bugs or feature requests.

It will be interesting to monitor how future development by departments is merged, or not, with the generic version after the end of the project. CARET will monitor this as part of our evaluation of the processes discussed here, including beyond the end of the project.

Specification-variability considerations

Where proposed software is required to accommodate large variability between 'branches' (such as in this case, between departments), the TODB development model scales elegantly and well: where variability is small, the development approach more closely resembles a more conventional software development project; and where variability is large, this approach allows each instance to be quite different and for the differences to be pursued independently. This makes it a powerful model. Centrally hosted and supported software has its advantages, however, and the TODB development model could be changed to a conventional centralised model if it transpired that inter-stakeholder differences are very small. The only danger of such a decision might be that the architecture of the system might be inappropriate – as software designed to be maintained by 'non experts', and so free of abstractions, use of 3rd-party libraries and so on, might, perversely, be less efficiently maintained by 'professionals'.

Inter-instance compatibility

In general, it does not matter what each department does with its instance of TODB after release. The one caveat to this is with the benefits that would accrue to have TODBs communicate with each other. Being able to view the activities of a staff member in another department (in a shared lecturing environment) can be extremely valuable. There is a danger that this communication mechanism could break down through uncoordinated changes.