https://doi.org/10.48009/2_iis_2007_529-533

CREATING PLUGGABLE DOMAIN-PLATFORMS FOR GOVERNMENTAL SYSTEMS

George Sargent, University of Wisconsin-Whitewater, sargentg@uww.edu
David Munro, University of Wisconsin-Whitewater, munrod@uww.edu

ABSTRACT

Governmental entities, by their nature, repeat geographically across the nation and around the world. As such, they have duplications of needs and of course this leads to "reinventing of the wheel". Given the available technologies available today, much of this duplication can be eliminated using the "open source" software paradigm for the construction of information systems. The goals of this paradigm shift are to (1) make the creation of governmental computer systems more efficient by reducing duplications, (2) spread the expense, (3) create better systems and (4) make the systems available to other government agencies.

Keywords: pluggable, open source, collaborative technologies, paradigm, international IS, government

INTRODUCTION

We are in a time of increasing needs for government information systems as too many of these systems are seriously flawed, or completely unusable. Just considering our own state, Wisconsin, the Milwaukee State Journal newspaper recently reported the state has recently scrapped two computer information systems projects after having spent $36 million and in recent years the total exceeds $100 million. [4].

A common governmental problem is paralysis by analysis. In an attempt to comprehensively address an issue they develop large, long-range plans. Such multi-year, software projects seldom succeed. For example, Wisconsin's Department of Health and Family Services has a web page for "County – State IT Collaboration".[10] On it you will find that in late January of 1997 the first deliverable was "a standard set of terms and definitions which describe the processes of our human service delivery system." Nearly seven years later, the agenda for a December 19, 2003 is for a "Planning meeting to discuss and agree upon a go-forward plan/project for achieving an integrated health and human services technology architecture." Seven years and still just talking!

Contrast this with SAHANA an open source project that was first conceived of during the December 2004 Asian Tsunami. SAHANA developed a system to aid Sri Lanka after the Tsunami. In 2005 the system was deployed in Pakistan to provide support after a severe earthquake. It was then deployed four more times in 2006 by various governments and NGOs. http://www.sahana.lk/ The approach taken from the onset was to provide an integrated set of pluggable, web based disaster management applications.

The handwriting is on the wall about the open source model and pluggable systems. This paper argues that putting these two together will yield much more successful systems at reduced costs.

OPEN SOURCE AND PLUGGABLE SYSTEMS IN GOVERNMENT AGENCIES

Open source systems have shown unquestionably good results. Eric Raymond in his Book "The Cathedral and the Bazaar" stated, "Who would have thought that a world class operating system could coalesce, … out of part-time hacking by several thousand developers scattered all over the planet", Raymond [8]. "Eric Raymond's persuasive manifesto defining the open source revolution has helped propel this collaborative approach to software development into the mainstream." [6] The Open source software development model is no less than a new software development paradigm.

The open source model, being collaborative in nature, is much better if used to do fewer projects shared by many people. Fortunately, this does fit the realities of governmental systems. There can be little argument that peer governmental agencies at all levels have overlapping problem domains whether we're talking about nations, states, regional or functional authorities. These overlaps foster "recreating the wheel" behaviors. Why not collaborate to design a single solution for a business domain that can be configured to meet multiple agencies' needs?

Fortunately, we also now have "pluggable" architectures for building systems that can be adapted by their users to meet their idiosyncratic needs. These adaptations can be made without rewriting the code base, or resorting to least-common-denominator solutions. In the past, rewriting parts of a system to adapt it to local needs stifled progress because any new release required rewriting the enhancements all over again. However, this is no longer true. We now can create new functionality and "plug" it into the standard system without changing the existing code base. When the standard system is enhanced, we plug our changes into the new release with minimal work.

Therefore, this paper argues that governments can greatly benefit from adopting these two advances in combination. That is, adopt the open source approach to building systems and then designing their systems so other developers can "plug-in" locally customized components to meet local needs.

BENEFITS

The first benefit of moving to the open source paradigm is increased participation and accountability. Many people can contribute – including people not on the project! The normal mechanism of becoming a "committer" to an open source project is to begin correcting bugs on their bug list. Such increased participation uncovers all sorts of software shortcomings. According to Raymond [7] "all bugs are shallow to someone." Many sets of eyes on the source code will discover not only bugs, but shortcomings early when they are least costly to repair.

Wouldn't states like to avoid headlines like "State Scraps computer project?" Apparently Raymond makes a compelling case for use of open source methodologies as he is often cited by people extolling its virtues, Whitlock [9], Porterfield [5].

A second benefit is to spread the costs and opportunities. If two agencies jointly developed a system, and were able to foster an open source community, they might get significant contributions from outsiders as do other open source projects. We might even tap into a sizable citizen group willing and capable to freely work on behalf of the government as they now already do for other open source projects! All of this extra help should reduce costs to each involved agency. Any subsequent agency that decided to consume an existing solution would save even more. When enough solutions were available, most every agency could also become consumers as well as producers.

Thus far, we've tried to make the case for the efficacy of the open source paradigm in general and the need for government entities to band together to use open source to reduce waste and cost. There is another important enabling technology.

Computer science has given us rich tooling and application frameworks. Of particular interest here are "pluggable" designs. The Eclipse editor is one notable example of a pluggable design. Tooling vendors regularly extend Eclipse by creating plug-ins to give it new functionality. (There are now over 800 plugins.) [2] Eclipse users then can install these without Eclipse even knowing of their existence and they work!

We want to capitalize on this pluggable design concept in a two step process. Create a pluggable skeleton of an application for a given business domain, then create multiple, e.g. two, pluggable applications based on the skeleton at two or more sites. The mechanism of ensuring the skeleton applications meet the needs of multiple agencies is to do it open source. Multi-year requirement gathering would be even more obviously unnecessary.

We might ask, is this concept realistic? How would we begin?

CONCEPTUALLY – HOW

The first step of creating a pluggable, domain skeleton is to use collaboration tools used by the open source communities such as Apache.org and Tigris.org. At minimum, these tools include web sites, discussion boards, to do lists, version control, bug tracking and a framework to support continuous builds, e.g. CruiseControl[3].

They also need to agree on their development platform. Let's assume they have agreed to build Web applications based on Java, a particular Java persistence model, JavaServer Faces and JavaServer Facelets (https://facelets.dev.java.net/), all selected for their pluggable capabilities.

By simultaneously developing a system for multiple agencies, each agency's design needs will get incorporated into the design early on. In the simplest case, this system is going to consist of two kinds of components which form the **domain platform**:

- A pluggable and therefore, reusable skeleton [1]
- Pluggable, fleshed-out framework applications built on the skeleton

Let's use the example to illustrate the difference between the skeleton and the fleshed-out framework. Suppose many states needed a new hunting/fishing license system and the project was begun collaboratively by several states and a Canadian providence as an open source project using the pluggable architecture concepts.

Our system would clearly maintain information about the hunter and the license assigned to the hunter. We could build a skeleton based on the process of taking input data about the hunter, validating it, and updating the data store to perform the usual CRUD (Create, Read, Update and Delete) on the hunter table. Every framework application will need this capability. The skeleton will perform these functions with no concept of any of the data elements contained in the hunter 'object'. It will only know to transmit, call a place-holder validator, store/retrieve and return the hunter object. Likewise the license will need similar functionality. The skeleton would be used unchanged from site to site.

Clearly implementation details will vary radically from site to site. Handling these is the job of the framework application. There will be multiple such framework applications because of site-specific needs.

These framework applications, unlike typical applications, would be designed to extend at logical extension points (discussed below) so that local customizations would not clash with future releases of the original code base. That way if a new release of the code base were made, the local interface implementation would not be affected so long as the method calls in the interface did not change. They would plug into the new release just like they plugged into the original release.

In the open source tradition, both parts of the application would be freely available to all.

We have the technological base to design applications that are robust at the user interface (UI) layer, business logic layer, and database layer so local needs can be met by extending the application with plug-ins without the fear that the next release is going to be a nightmare to install.

Just think as more states used our hunting/fishing license system, more people would discover and fix its shortcomings and, in turn, make the improvements available to all. This is a win-win proposition.

SPECIFIC PROBLEMS

Continuing with the example of developing a hunting and fishing registration system to support needs in Canada and the USA, assume two cross-border agencies are cooperating and they are using our suggested methodologies. Although they agree on the platform, they immediately determine they differ in fundamental ways:

- Language: English and French
- Measurement units: Imperial and Metric
- Monetary: Canadian and US dollars

After a little more discovery they determine that they also differ on:

- Desired look and feel of the application
- Database management products used: Oracle, MS SQLServer and DB2 (all three are used)
- Database structures, e.g. table names, fields. One site has more tables than the other.
- Custom validation needs
- Content needed by one, but not the other.
- Whole sub sections of unique content needed by one site, but not the other.

The above list of differences may seem daunting, but we now have technical solutions to all of them so that a user could make changes to support local needs and new releases of the software would not invalidate the local changes. Look at them one, by one.

Language: English and French. Java supports resource bundles. This allows the user interface (UI) developer to place a key value representing a place-holder for text on a display, and at run-time the actual text is retrieved from a file, i.e. resource bundle. Consequently, support for the new language is largely handled by making a parallel file in the new language.

Measurement units: Imperial and Metric. Java supports the Internationalization standard, I18N and localization, L10N. These reconfigure measurement units, decimal points, error messages and so forth to the selected language as configured. (Clearly, it could also convert units of length or volume without difficulty if that were a requirement.)

Monetary: Canadian and US dollars. The application can take care of details involving conversion, e.g. between US dollars Canadian dollars in our example, if any conversion is even necessary.

Desired look and feel. Style sheets are great for defining new looks including the location of the various components on the display, e.g. locate on right side, or top. Furthermore, Java Swing allows various looks & feels to be chosen for each application.

Database management product used. Here we are getting into more serious changes. However, the Java persistence layer will work with any database, e.g. Oracle or DB2. Any customization can be carefully separated from the domain platform such that it is not accidentally lost on subsequent updates of the domain platform.

Database structures. Here the problem is that everyone's databases already exists and are different so the application has to be fitted to each database.

The application can use the concept of "convention over configuration", popularized by Ruby on Rails, to minimize the configuration tasks. This concept has caught on like wild fire and should simplify database adapters. However, it will not solve all the data differences problems. Local applications will need further adaptations. However, local data handling needs could be isolated in the application's data layer to protect them from future releases of the platform.

In the end, each locally customized application will need to write some of its own SQL to perform CRUD operations on its own data structures.

Custom validation needs. The validation process is always the same, but the implementation details vary. The process always takes a value, or set of values in, evaluates it/them and returns true/false to its container which branches according to rules implemented in a configuration file. Template files can provide things like maximum and minimum values and where to branch given the result.

But validation requirements may vary much more radically. Suppose one local implementation may want to examine driving records for violations before issuing a hunting license. That requirement is not shared by the other sites. Nevertheless, the extension point will be provided in the platform to do it. One such mechanism is the "dependency injection". See http://en.wikipedia.org/wiki/Dependency_injection. This will give each local implementation the power to have extensive, customizable business logic validation to meet site specific requirements without changing the code base.

Likewise, an error message area will be provided so each validation could provide custom feedback.

Content needed by only one site. Assuming these content differences are minor, one site will add new fields to the presentation to placing them in a separate file and "including" it in the original page. This way the modified content is not mingled with the framework application and updates are not impeded greatly. Each field specifies its own validator, so the server-side validation may be also specified unique to the newly added field. But alterations may not be so simple.

Whole sub sections of unique content. Perhaps this is the site that wants to examine the driving record before issuing the license. Assuming the presentation is built with JavaServer Facelets, the page is divided into sections or facelets, e.g. top, bottom, left, right, center, etc. Here the site needing radical changed would create a new facelet and either replace some other facelet or add it to the existing presentation. That's the beauty of facelets. The applicant's drivers record may be pulled up onto a facelet to give the site a custom capability.

CONCLUSION

We have shown that the Information Technology industry has given us tools that may be used by governmental agencies to radically alter the process of developing software to achieve better results. These tools are mostly available free and are consistent with the needs of governmental agencies. We think that if used properly, governments could radically alter the way software is developed for the betterment of all citizens.

REFERENCES

1. A Pluggable architecture, e.g. Java Plug-in Framework: JPF http://jpf.sourceforge.net/)
2. Count of available plugins for Eclipse, http://www.eclipseplugincentral.com/.
3. CruiseControl, http://cruisecontrol.sourceforge.net/
4. JSOnline, Milwaukee Journal Sentinel, *State scraps computer project*, Feb 17, 2007
5. Porterfield, Keith W, *"Information Wants to be Valuable", O'Reilly Perl Conference*
6. Raymond, Eric S, (2000), *The Cathedral and the Bazaar*, http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html#catbmain
7. Raymond, Eric S, (2000), Release Early, Release Often, Principle 8
8. Raymond, Eric S. (2001), *The Cathedral and the Bazaar*, back cover, O'Reilly, 0-596-00108-8
9. Whitlock, Natalie, (2001) *The security implications of open source software*, IBM Developerworks, [online] http://www-128.ibm.com/developerworks/linux/library/l-oss.html
10. Wisconsin Dept. Health and Family Services page http://dhfs.wisconsin.gov/aboutdhfs/ITcollaboration/