The “Swim” System for User-Oriented Presentation of Test-Case Results

Ward Cunningham, AboutUs
Bjorn Freeman-Benson, Eclipse Foundation
Karl Matthias, Eclipse Foundation

Introduction

As staff members of the Eclipse Foundation, we faced the challenge of automating many of our manual workflows while minimizing the cost of doing so. Having seen similar efforts fail, we chose a new design point that we believed would entice more of our stakeholders to participate. This paper is about that design point and its success-to-date.

The Eclipse Foundation is a not-for-profit, member-supported corporation that hosts the Eclipse Projects and helps cultivate both an open source community and an ecosystem of complementary products and services. In general, the Eclipse Foundation provides four services to the Eclipse community: IT infrastructure, IP management, project management, and ecosystem development. Many of these services contain processes best described by an event oriented workflow, for example, the election of a new Committer to an Eclipse Project (http://tinyurl.com/2bxnyh) has events such as “start election,” “vote,” “approval needed,” “committer legal agreement received,” and so on. A simpler example is changing a member's address and phone number (http://tinyurl.com/22rqdg): even this seemingly simple process involves checking if the address change is triggered by an employment change because an employment change will require new committer legal agreements.

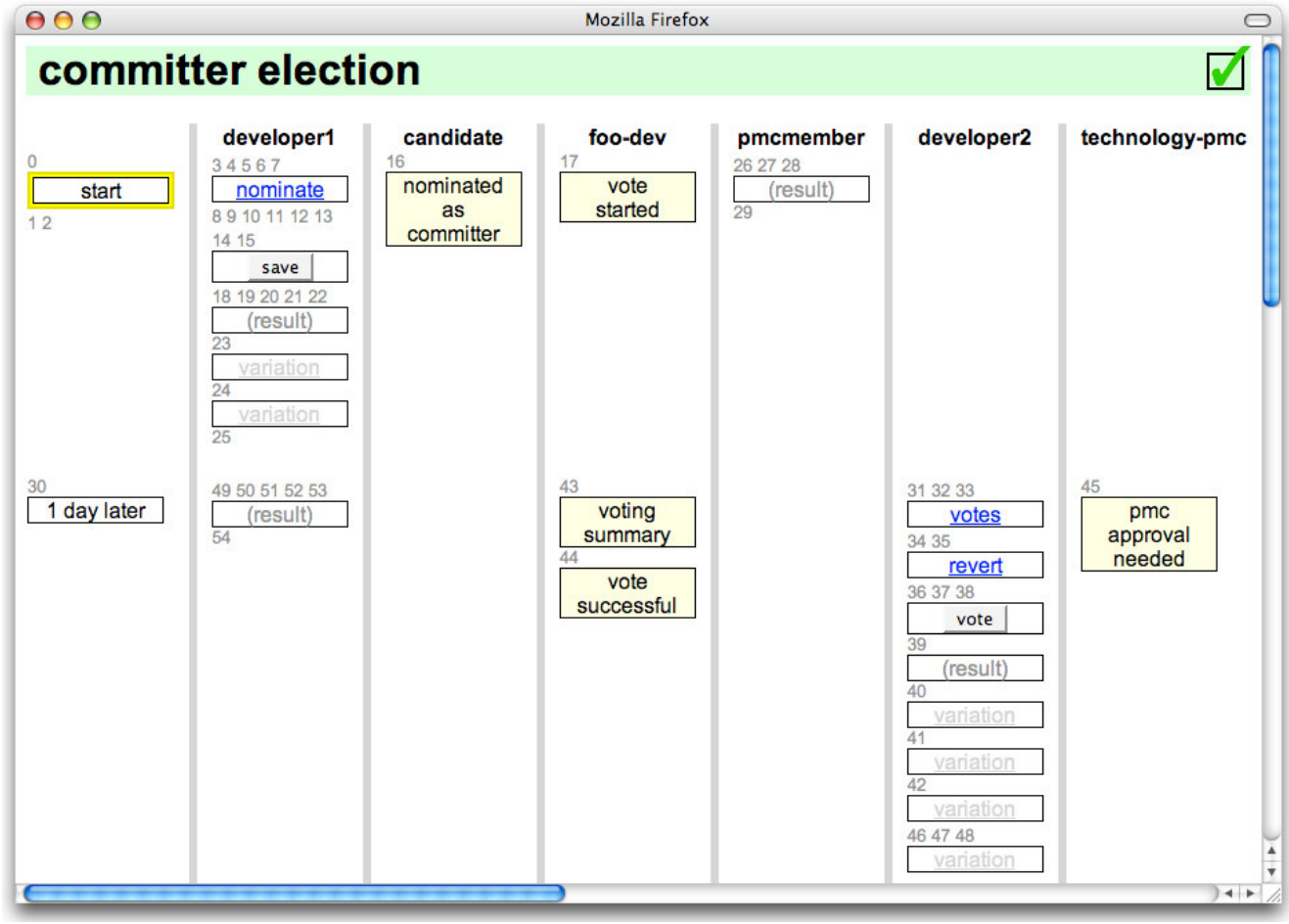
Like all organizations, we don't have resources to waste, so an important characteristic of our solution was our investment in durable test scripts: we didn't want to write the scripts once, use them to test the system, roll it to production, but then never use the scripts again. We believe that the only way to protect our test script investment was to involve all the stakeholders in the production and consumption of scripts, and the only way to achieve *that* was to use the right level of abstraction in the scripts (and thus not try to do detailed testing of every API).

Visualizing Long Running Transactions

Our realization, and thus our solution, is based on the fact that the Eclipse Foundation workflows are really just long running transactions. The transactions operate against a database (technically against five separate databases, but that's an almost irrelevant technical detail) and our user interface is HTML and AJAX. The code is all written in PHP.

Our testing solution is to script and visualize these long running transactions by simulating them using test databases and capturing the output for display.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{screenshot of committer election swim diagram}

The simulation uses as much of the real application code as possible, replacing only the real databases with test databases, the user input routines with the test scripts, and the browser output with output stored in buffers for post-processing and evaluation. Consequently the simulated transactions are the real transactions and the scripts are testing the real application code and business logic.

Because our transactions can take weeks or months to complete, we obviously run our simulations faster than real time using a simulated clock. The visualizer post-processes the results of the simulation to produce an overview two dimensional picture of the progression of the transaction through time, using a page format inspired by the "swim lane diagrams" made popular in business process reengineering [Rummler & Brache, 1995]. The overview contains a column of each person involved in the transaction and icons with fly-out details representing important interactions. This format has worked well for our need, but could easily be substituted with some other format without invalidating the rest of our results or even modification of the test scripts.

The simulator and visualizer are used in development and testing, but are also deployed with the finished application as a form of documentation. This use of *the tests as documentation* has the natural advantage of the documentation being automatically maintained, but it also forces us to make design choices so that the tests actually make good documentation. In our previous

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

attempts to provide this same “the tests are the documentation” failed because the tests contained too much detail to be good documentation; detail that was necessary for testing, but overwhelming as documentation in the sense of “can't see the forest for the trees”. Our solution here has a carefully chose set of abstractions that make the scripts both useful as tests and useful as documentation.

Visualizing for Programming and Testing

The swim visualization we have implemented is useful for both programming and testing. We take as a given that programmers benefit from test driven development. We also take as a given that the agile style of “all programming is extending a working system” is an ideal way to build systems in partnership with users.

Programming

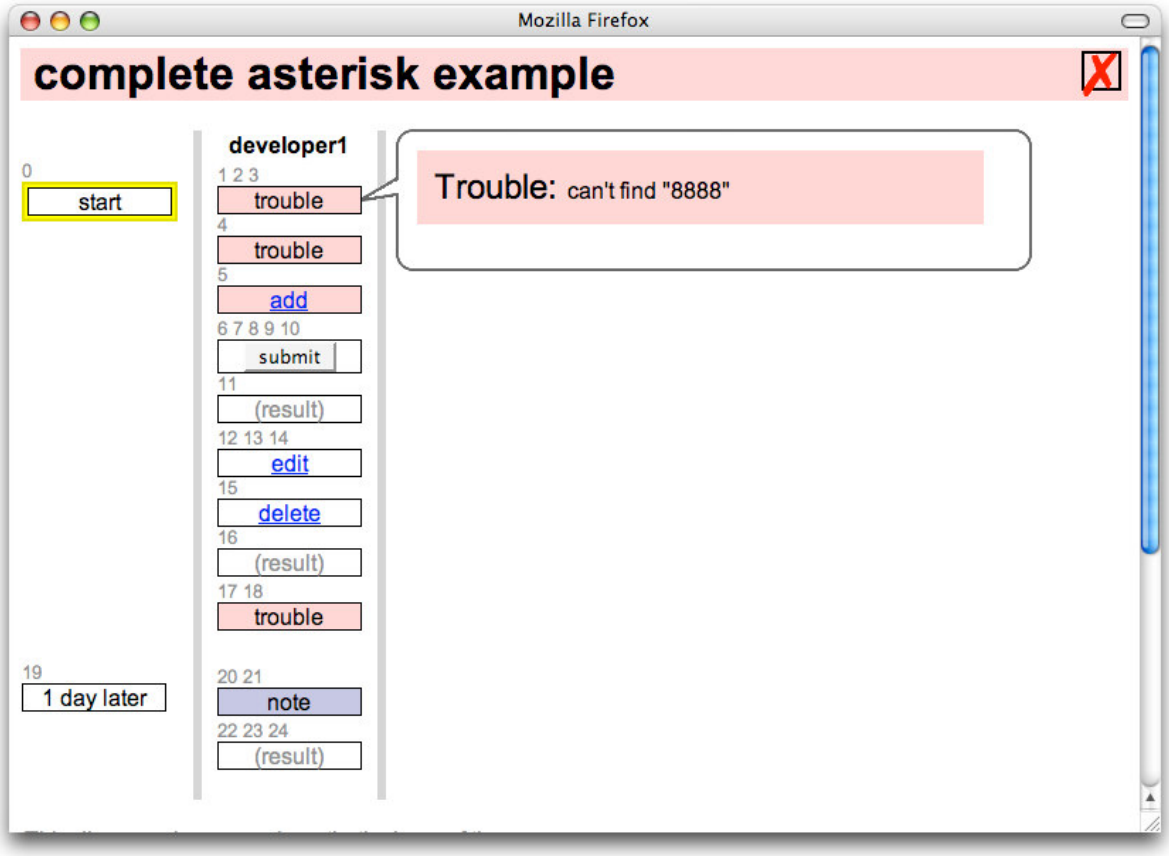
Our simulation and visualization system supports these programming styles in a number of ways. First, it provides the real “as is” system for detailed study by the programmers and testers. Second, the test scripts are just PHP code and do not require any special software or training, thus they are easily composed in advance of programming.

```
1: login('developer1');
2: find('asterisk_manager');
3: check('8888');
4: check('60 minutes');
5: press('add');
6: enter('pin', '4321');
7: enter('start', '2009-02-20 00:00:00');
8: enter('stop', '3');
9: enter('description', 'test conference');
10: press('submit');
11: show();
12: check('[add]');
13: check('8889');
14: press('edit', '8889');
15: press('delete', '8889');
16: show();
17: omit('54321');
18: check('8888');
19: advance('1 day');
20: login('developer1');
21: note('Old conferences are removed');
22: find('asterisk_manager');
23: omit('8888');
24: show();
```

{example test script}

During development, a test failure may result from incorrect application code or an incorrect test case. These failures are rendered with red boxes within the visualization, allowing the programmers to narrow their search for the cause.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{visualization of failed test}

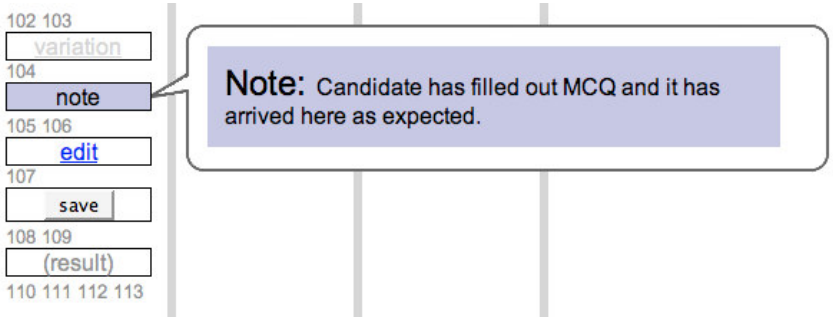
One of the features we've added to our scripts is the option to verify whether all of the output has been checked against expected values. Our scripts are often arranged in suites with one main comprehensive test and a bevy of smaller variations. The main test verifies all the output against expected values; additionally, because it verifies all the output, it fails if there is additional unexpected output.

Additional smaller test scripts only verify the portions of the output that are new and/or different from the main test and do not re-verify previous results.

{ignored emails at end of test}

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

One programming technique that we have found useful is to add "notes" to a script under development, often pending changes that we haven't gotten around to including yet. Other uses of the notes include describing interactions between the user and the system that are not captured by the script.



{notes used for out-of-band data}

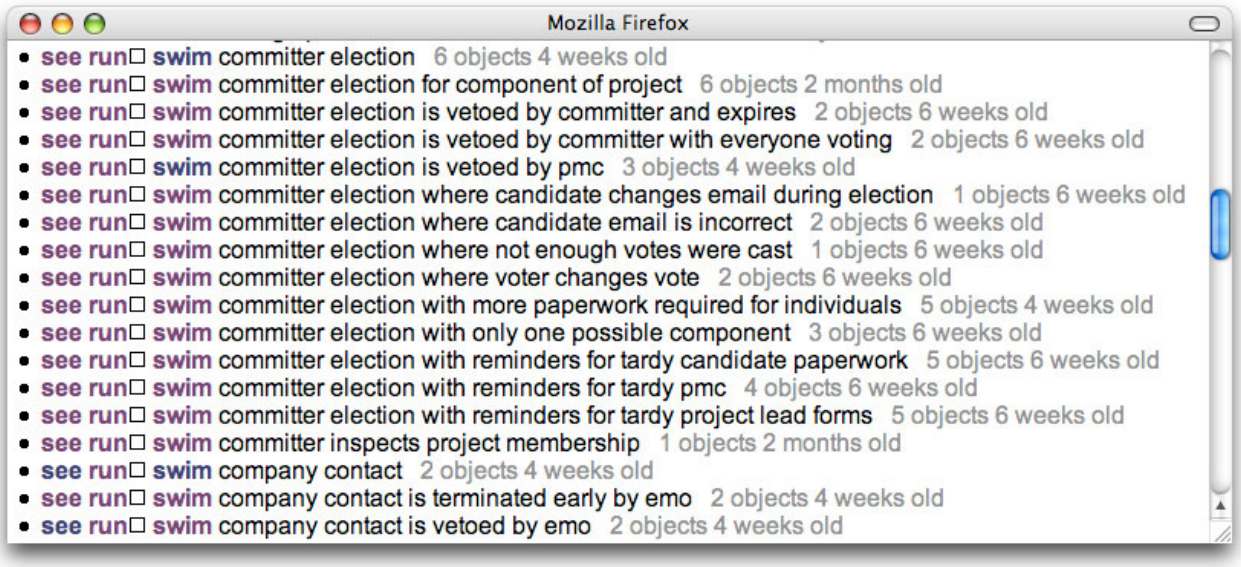
Another very useful programming technique is annotating a script with "variations". Variations allow us to group scripts without resorting to external meta-data. Variations are noted in the script in a straightforward manner identical to all other script commands. Variations are assertions that succeed if there is another script with the variant name, e.g., if a script named "contact address" has a variation named "two quick changes", then the variation command will succeed if there is another script named "contact address with two quick changes".

{variations visualized in swim diagram}

```
38: press('vote');
39: show();
40: variation( 'voter changes vote' );
41: variation( 'vetoed by committer and expires' );
42: variation( 'vetoed by committer with everyone voting' );
43: email( "+1 for Karl Candidate", "foo-dev" );
44: email( "Voting is complete", "foo-dev" );
45: email ("PMC approval needed", "technology-pmc");
```

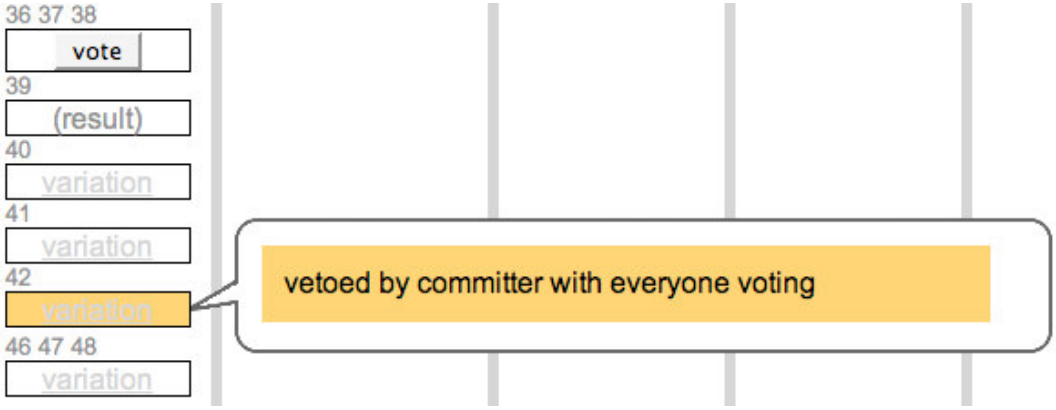
{variations in the script itself}

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{list of scripts with the variant names}

If the variant script is missing, it can be in one of two states: either not yet written, or under development but not yet completed. If the first case, the variation command always causes the current script to fail, thus prompting the programmer to write those variant use case/test case scripts. In the second case, the variation command will succeed with a warning on a development machine, but will fail in the deployment test. The goal here is to allow the programmer to continue development (all written tests pass with green), but not allow her to deploy a production system that is potentially flawed due to missing variations.



{variation that is yellow indicating "not complete"}

The feel of the system is the same as large unit test suites: due to the heavy use of databases and the need to reinitialize the databases to a known state for each test, the tests run somewhat more slowly than code-only unit tests. However, the system is fast enough that every developer can run all the tests before releasing new code.

We have found that the tests are easy to write and flow naturally from discussion with the stakeholders. Our informal impression is that we spend about 10% of our time writing test scripts as opposed to the 50% we used to spend when we wrote more detailed unit tests. We

attribute this difference to a difference in testing philosophy: these scripts are *use cases* rather than *unit tests*. As use cases, they are demonstrating and documenting a particular long duration transaction. Because the scripts and visualizations are easy to write and read, we get good cooperation from the stakeholders and that leads to our lower effort. Additionally, because the scripts are not unit tests, they are not rigorous tests of the APIs and thus do not carry adverse properties that such tests entail, such as brittleness.

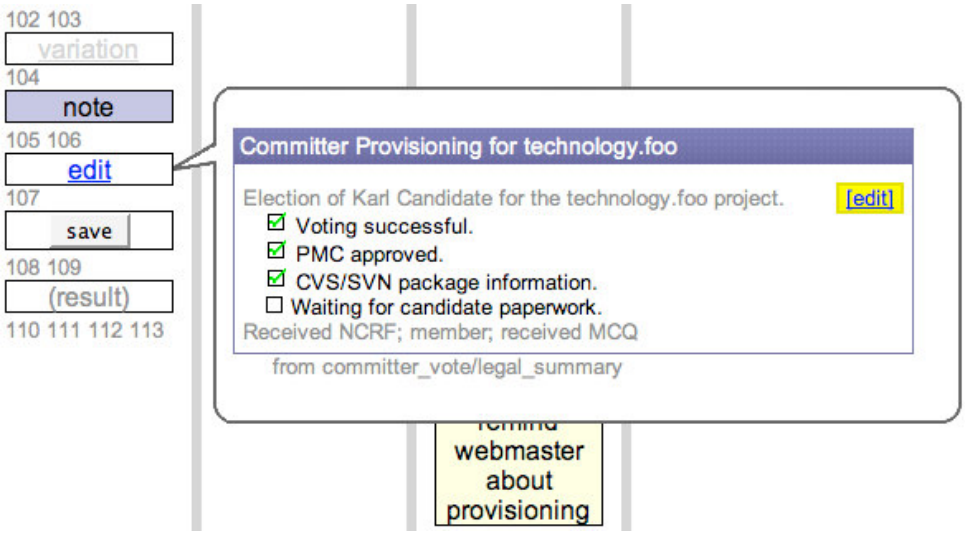
We do write scripts for both sunny and rainy day tests for the underlying business processes; we understand that sunny day tests alone are completely insufficient for a robust system. However, our rainy day tests are tests of the business logic, not of the underlying frameworks, PHP engine, MySQL databases, or Apache web server – we accept the low risk of those untested failure modes as appropriate to our mission. Our IT staff provides additional up-time testing of those services as a supplement to the scripts and visualization we describe here.

Testing

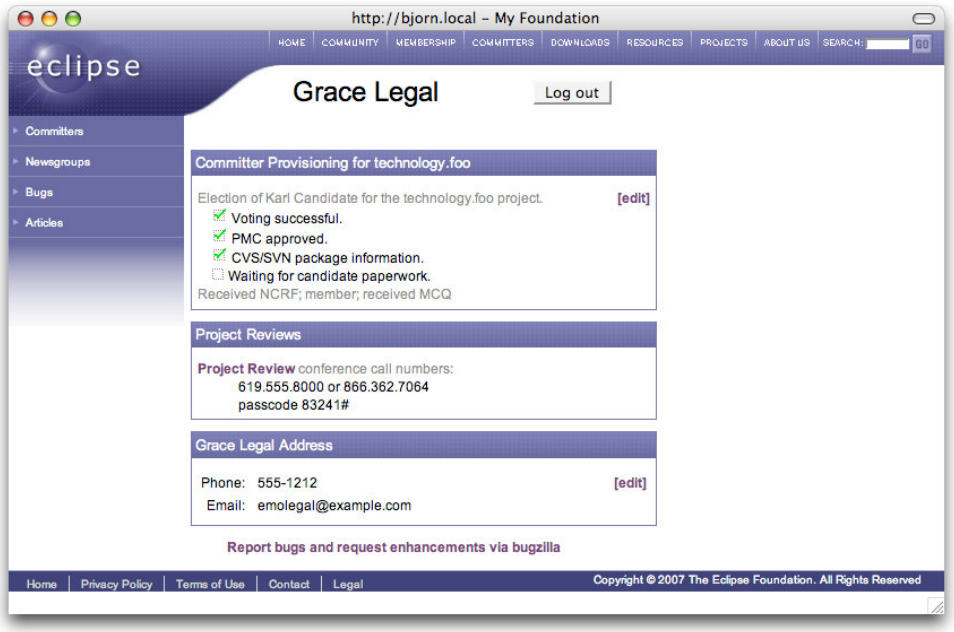
We believe that testers benefit from exploratory testing and thus our scripting and visualization system supports that exploration in two ways. First, the visualization works as a map of useful program and database states. Testers can use the map to direct their exploration and testing – for example, we find ourselves pointing at an area of a large diagram as we discuss working "over there".

Second, we designed the system to allow a user to switch from a passive examiner to an active participant in testing with a single click. First, all state variables are stored in the databases: we have a stateless web server without session states. Thus each step of the long running transaction results in a particular database state (set of tables, rows, and values). As we all know, setting up a particular complex state so as to replicate an interesting behavior or bug can be time consuming. The scripts, however, do that set-up automatically just by their execution. Thus we've implemented a feature where the user can click on any numbered step in a large diagram and be switched to an interactive version of the system (i.e. the real system connected to a test database), pre-populated with the database in the exact state that it was in at that step of the script.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{user about to click on the gray step number 105}

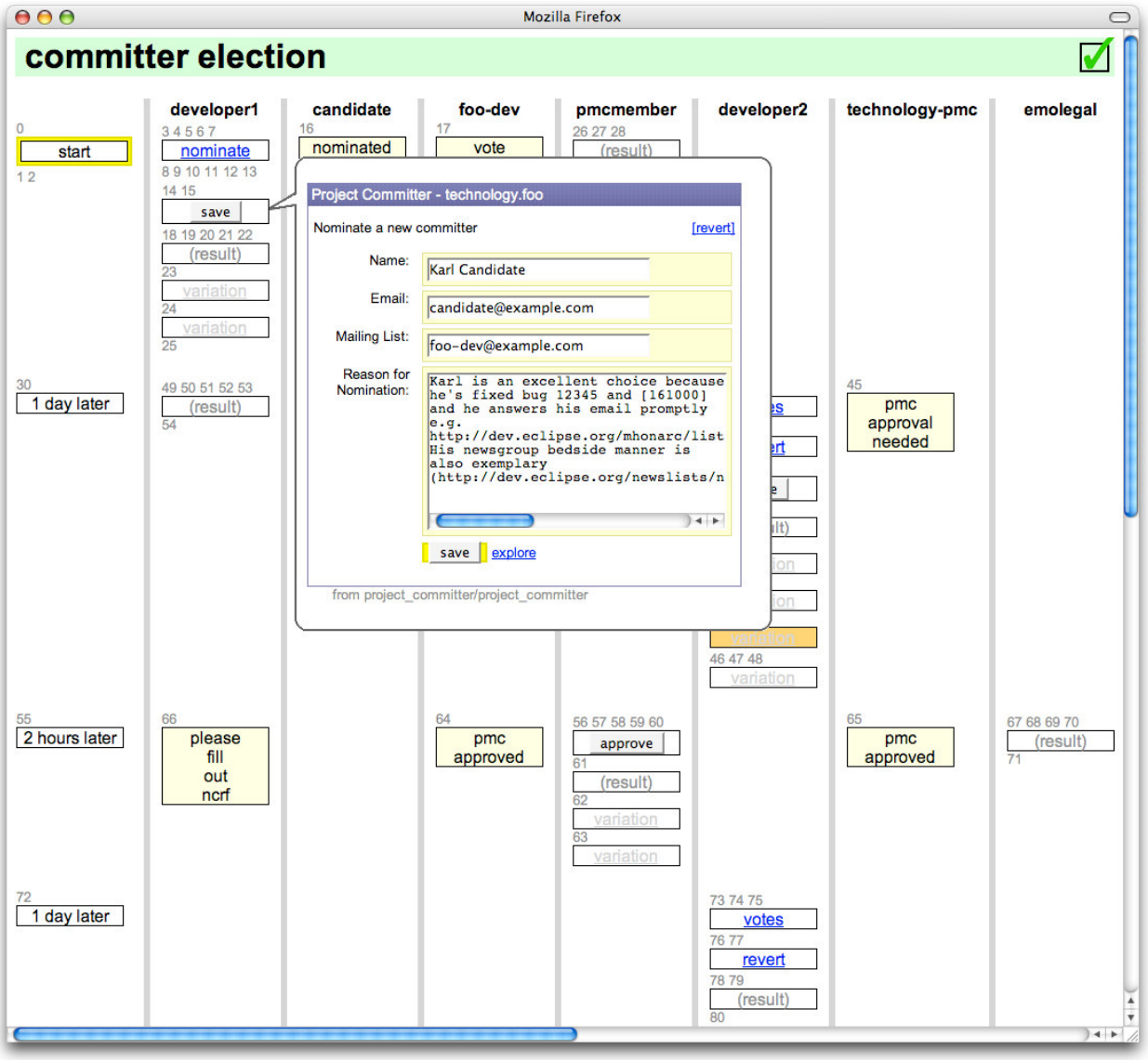


{user is taken to interactive interface with database initialized to step 105}

Visualizing for Conversations

The scripting and visualization system facilitates conversations with our stakeholders (analysts and users) in a number of ways. For example, the visualization of each script produces a unique URL, so of course that URL can be emailed and IM'ed to someone else. Additionally, the visualization is fairly generic HTML with CSS, so it too can be saved and emailed as a conversation reference.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

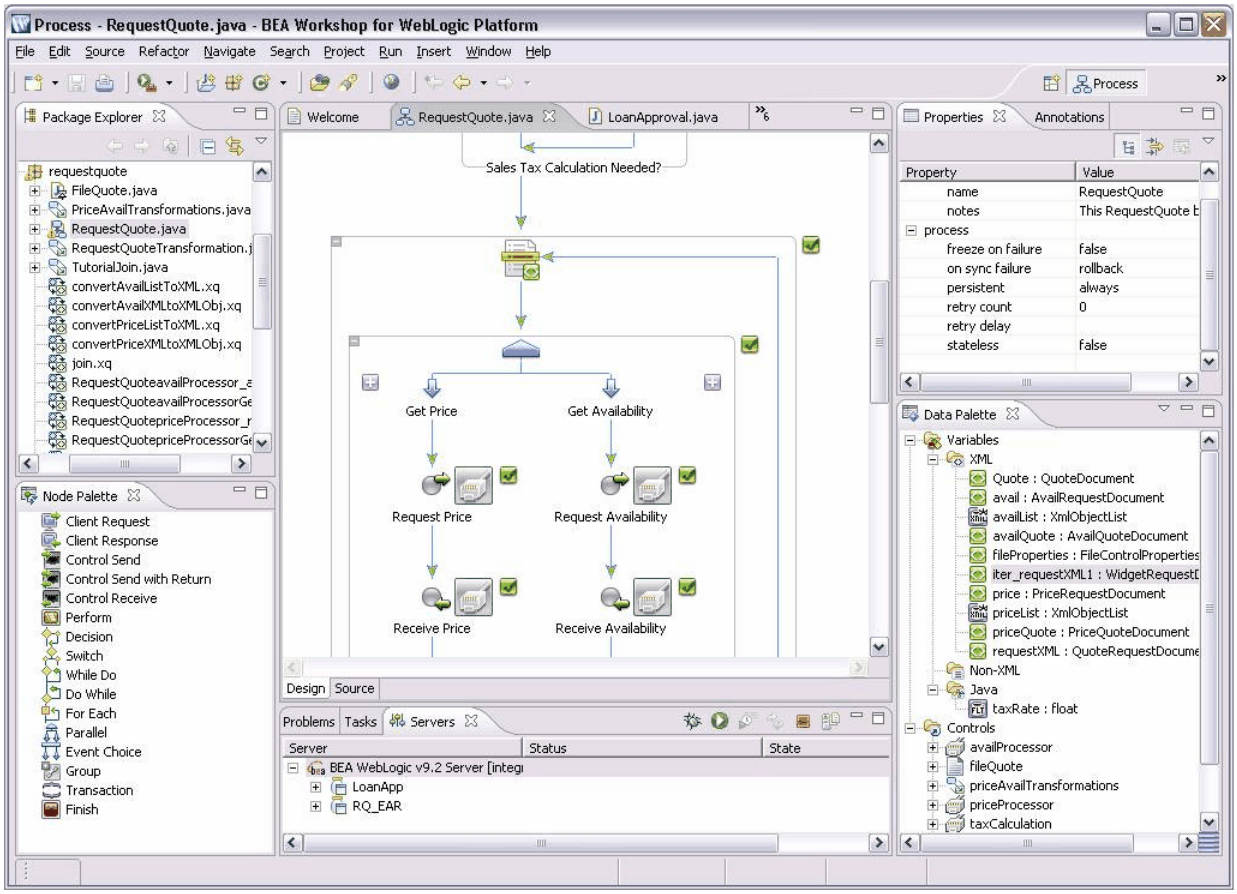


{general appearance of browser viewing a large swim diagram}

Each visualization has numbered steps which allow precise discussion even over the phone. We have already experienced a number of conversations along the lines of "after 46 and before 49, we need to send an email to the user informing them about X" or "here at 32, the dialog box is confusing".

Visualizations also have exact renderings of the real interface panels with soft yellow highlighting to show which fields and buttons have been modified and pressed. Because the state of the long running transactions are immediately visible in the interface panels, and because the interface panels are familiar to the users, we find that conversations with the users and stakeholders are much easier than if we had used a more traditional "bubbles and arrows" state machine description.

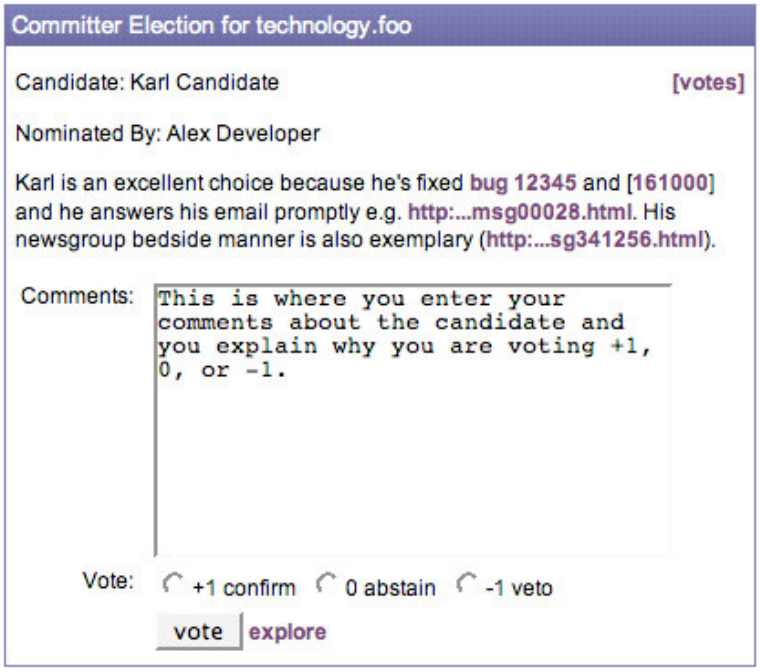
First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{general appearance of a BPEL bubbles and arrows process description courtesy http://e-docs.bea.com/wli/docs92/overview/}

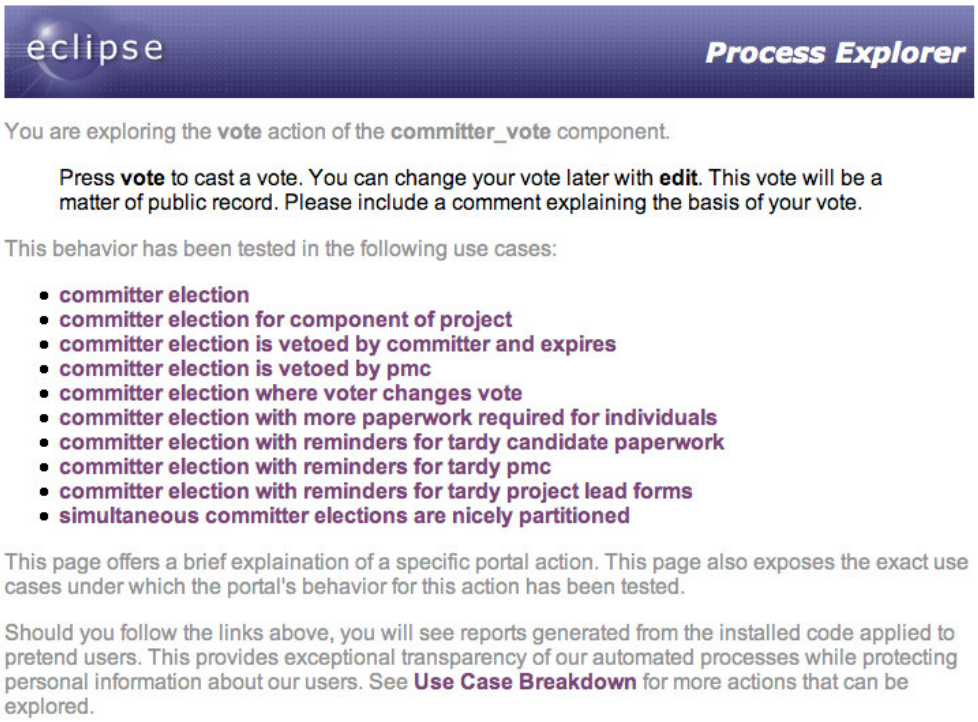
As we mentioned earlier, the simulation and visualization system is also available in production as a form of end-user documentation of the processes (long running transactions) being automated. We invite our users to explore these visualizations through an "explore" link by each action button in the user interface panels.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{explore link next to vote button}

The explore link implements the usual context sensitive help text one would find in an application, but it also provides additional context by showing the user all the use-cases containing that same action.



{process explorer text and list of use-cases}

We have found this additional context particularly useful because it helps explain the possible branches in the long running transaction, not just the single action (e.g., "save") that the user is about to perform. This is similar to the "see also" links in help systems, but because our list is the complete list of use-cases that define the state space, we believe that it more useful than a set of more general help messages.

Additionally, because we use the use-cases as the definition of, and the quality assurance tests of, the application *and* then the users use the same use-cases as the documentation of the application, both ourselves and our users "know" the application through the same set of use-cases. This common set of vocabulary (use-cases) definitely facilitates conversations and reduces common "programmer versus user" misunderstandings.

Open and Transparent

Because we are an organization dedicated to open source, we believe that both our code and our processes should be open and transparent. This visualization tool supports that belief that showing *all* states and sub-states of each long running transaction, even those that some other organizations might consider privileged. For example, even through our portal provides an interface panel for an Eclipse member company to change their contact people, a self-service activity that one might assume directly updates the database, the use-cases show that such changes require approval from Eclipse Foundation staff.

{swim diagram revealing that organization contact changes require approval}

Compare this transparency to submitting a review to a website like Amazon: you are never sure if your comments require approval by a human before posting, or perhaps are subject to retroactive disapproval by a human, or are never considered by a human at all.

An example of where this transparency was useful is that EG, an Eclipse Committer and leader, received an email from the portal and quickly emailed back to us "why am I receiving this?" We pointed him to the url defining the committer election process and the particular email he received and thus explained why he received that email. He examined the visualization and understood his role in the process.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

Benefits of Conversations

Our hope is that making the use-cases (the tests) available through "explore" links and our visualization system will make them valuable to more people. It simplifies or avoids the need to write extensive textual documentation due to the "a picture is worth a thousand words" characteristics of the visualizations and the fact that the comprehensive tests cover the expected user experiences.

Additionally, because the use-cases are both our documentation and our quality tests, we find that there is additional incentive for keeping the tests up-to-date.

Implementation

Our system is implemented entirely in PHP 5 and MySQL 5 on top of an Apache 2.x web server. It is operating system agnostic and has been run on Linux, Mac OS, and Windows XP and Vista. The code is released under the Eclipse Public License v1.0 [Cunningham, Freeman-Benson & Matthias 2007].

The system is stateless in the sense that each http request is processed without session state and thus all object state information is saved in the database through direct SQL calls. Each interface panel is defined by a separate PHP application object that processes method invocations and returns HTML.

The live system uses AJAX calls from the interface panels through a single point of control, dispatch.php, which verifies permissions, instantiates objects, invokes the appropriate action method, and then returns the new HTML. On the client (web browser side), the AJAX call replaces the old contents of the interface panel with the new returned HTML.

We chose to use straight HTML (or rather, HTML + CSS) rather than an object transport format such as JSON because we wanted all the rendering and layout to happen on the server-side. The option of moving objects across the wire and then rendering them on the client-side would have required that our visualization framework (which runs entirely on the server) exactly duplicate that client-side rendering. Thus, for simplicity, we chose to transport HTML and use the existing rendering functions built into the browser.

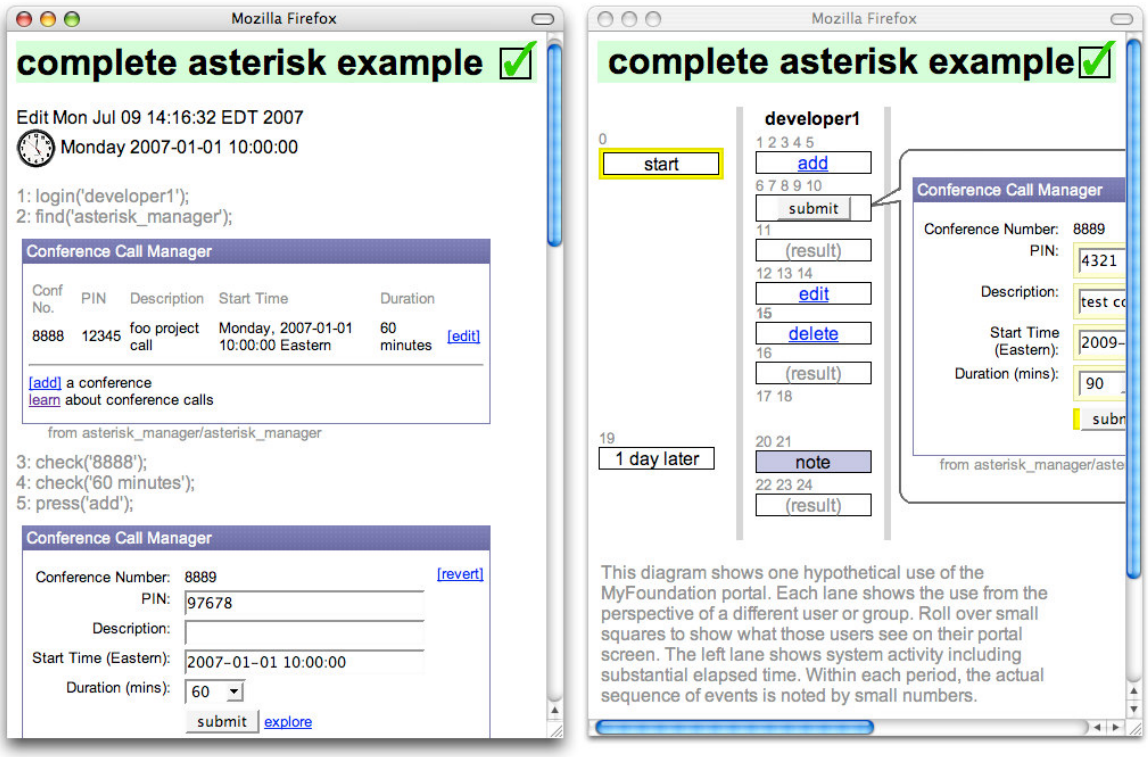
All of the context information, such as the names and logins to the databases, are defined in a single context object. This allows the simulator to easily replace all the live data with simulated data, and the real clock time with simulated clock time, etc.

In the simulation and visualization system, we replace the single point of interaction, dispatch.php, with a different control script, swim.php.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

Running a Script

However, before describing swim.php, we'll describe an intermediate step: run.php.



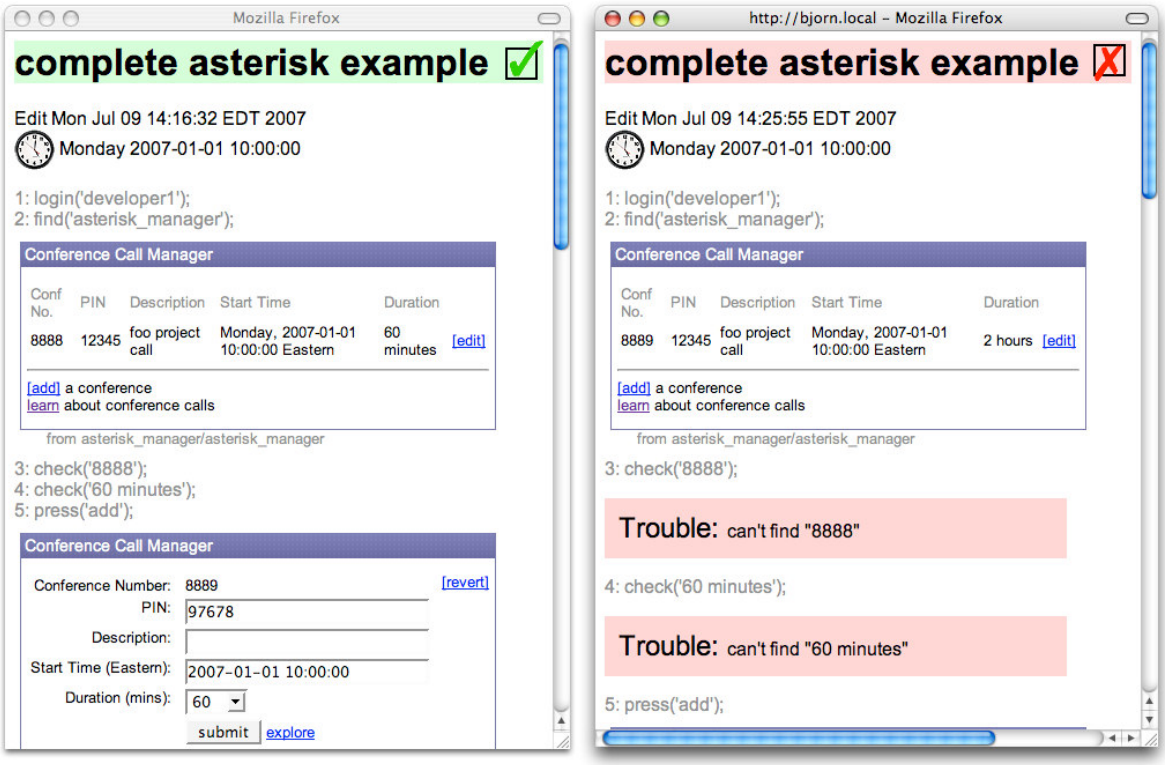
{comparison of linear run.php output (left) and columnar swim.php output (right)}

The run.php simulation and visualization is a simpler system than the swim.php simulation and visualization. Run.php was our initial version and is still useful for certain kinds of debugging. Run.php and swim.php use the same driver and differ only in the visualizer.

Our test scripts are just PHP, i.e., not some special testing language, and the test driver is effectively a PHP interpreter written in PHP. It reads and eval's the test script line by line. The test script is written as a sequence of PHP function calls to functions defined in our test harness.php file. These harness.php functions form a *domain specific language* for testing long running transactions and include: login as a particular user, enter some text in a form field, press a form button, assert that a string does/does not appear on the page, assert that an email was/was not sent, etc.

These harness.php functions simulate the AJAX actions that a user would trigger in using the live version of the system. Thus application object performs its real business logic (against the test databases) and then returns the appropriate HTML. Run.php displays the test script line numbers, the function being eval'd, and the resulting HTML all in a sequence on the output web page. We added a little Javascript using HTML entity ids and changing style sheets so that we can annotate the top of the web page (rendered early) with the test results (determined late). We also have a Javascript timer that catch the cases where the entire PHP system crashes so that we can also annotate those web pages as having failed.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

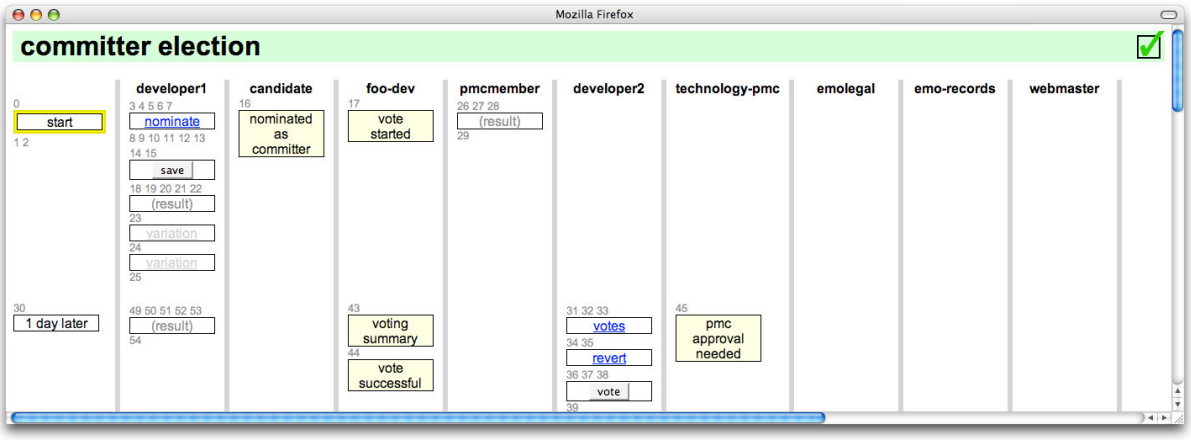


{comparison of successful run (left) and failed run (right)}

Visualizing a Script

To produce our two-dimensional visualizations, we replace the "print resulting HTML" of the run.php script with saving the resulting HTML from each step in an array indexed by simulated user and time. We then post process this array into an HTML table and print that table to the web page at the end of the simulation.

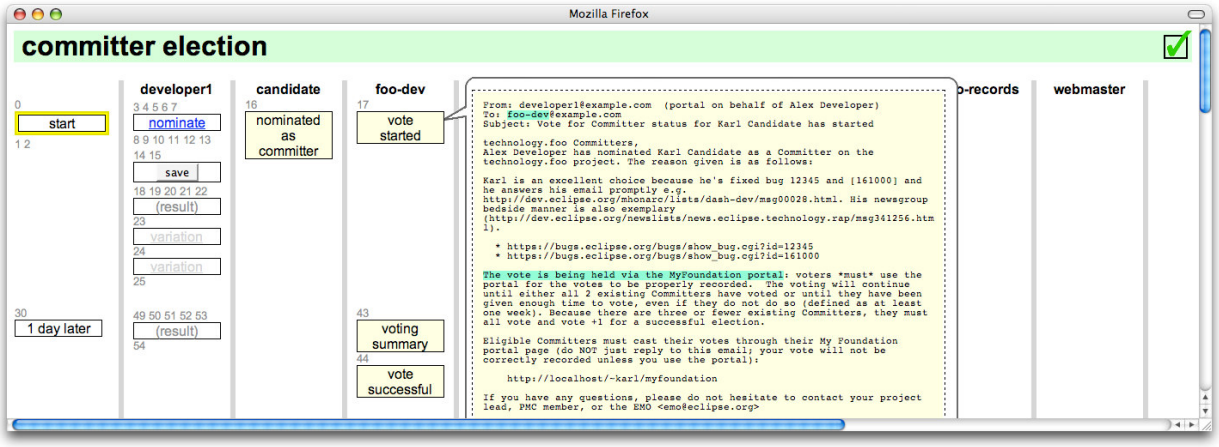
The columns of the table are the personas involved in the long running transaction. Some of these are the simulated users defined by the login(...) statements in the script. Other personas are defined by the application logic itself: for example, the committer election process sends notification emails to a number of interested parties who may or may not be represented by login(...) statements in this particular script.



{personas involved in committer election appear across top of output}

Obviously, the set of personas is not known until the script is complete and all the business logic has been executed and thus the visualization cannot be produced incrementally.

Application objects can report the *effective user* when their action is better classified under some other column. The best example of this is that emails show up under the "sent to" person, rather than the user who was logged in at the time the email was sent.

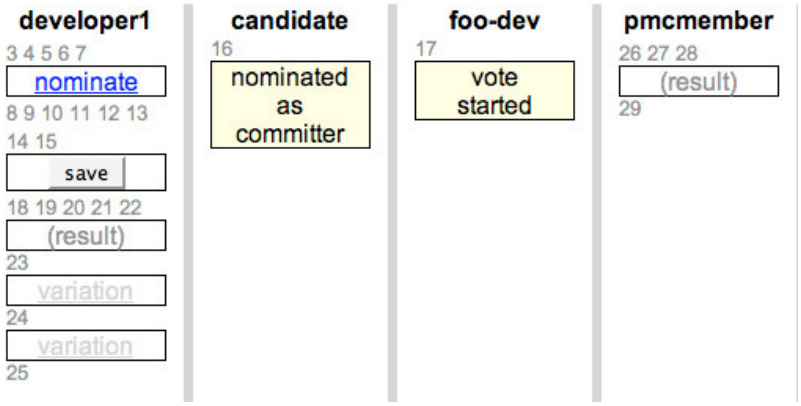


{email appears under sent-to persona}

We design our test data so that the personas are useful, e.g., "Polly Programmer" rather than "person1". We could use real users from our real production databases, but we choose not to due to privacy concerns.

The rows of the table are determined by the "advance" of time. We use an explicit advance(...) statement in our scripts to advance simulated time. There are two good reasons for this: first, by allowing many sequential events to appear in parallel in the table, we can have more compact table and, in spite of larger and larger displays, screen real estate is always at a premium. Second, it allows the script author to distinguish between important and accidental sequencing of events. For example, if there are four people voting in a particular committer election, the order in which they vote is not important and thus we show them as all voting in the same horizontal row.

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{one slice of time showing many things happening at once}

Because we use a real MySQL database for data and states, and because we use real DATETIME columns in that database, our simulated time has to produce real DATETIME values for the database at the same time it provides sortable values for the PHP code. We solved this problem by assembling a SQL expression for "now", and then always using the MySQL database to compare and resolve time variables.

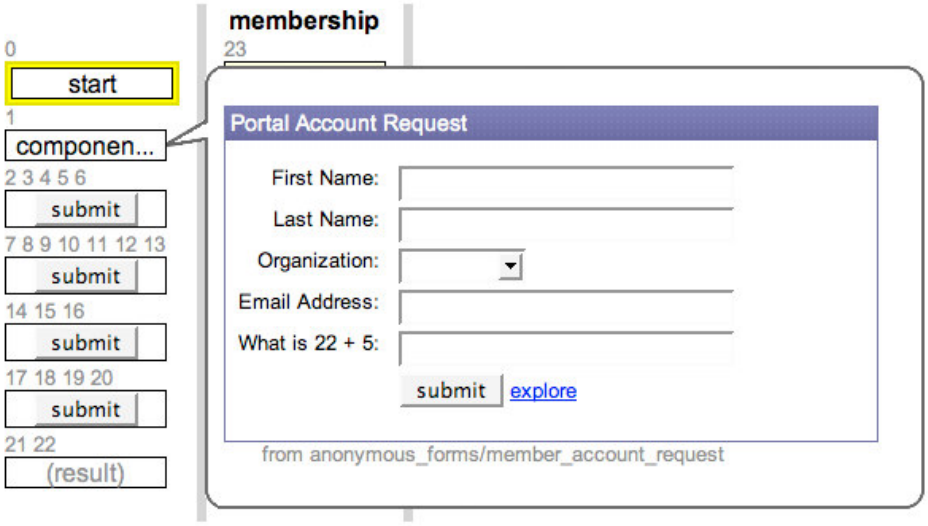
Visualizing an Action

Each action in the script is summarized as either just a reference number, or as a reference number and an icon. We defined a simple set of rules to determine when an action is interesting enough to merit an icon for example, press() and enter() statements are interesting, whereas login() and check() are not. The test script can override these rules with a show() statement, although we have observed that two additional rules would eliminate our need for show() statement.
These are:

- A login()/find() pair with no subsequent press(), is interesting and thus earns an icon.
- The last button press in a time segment in a column is also interesting.

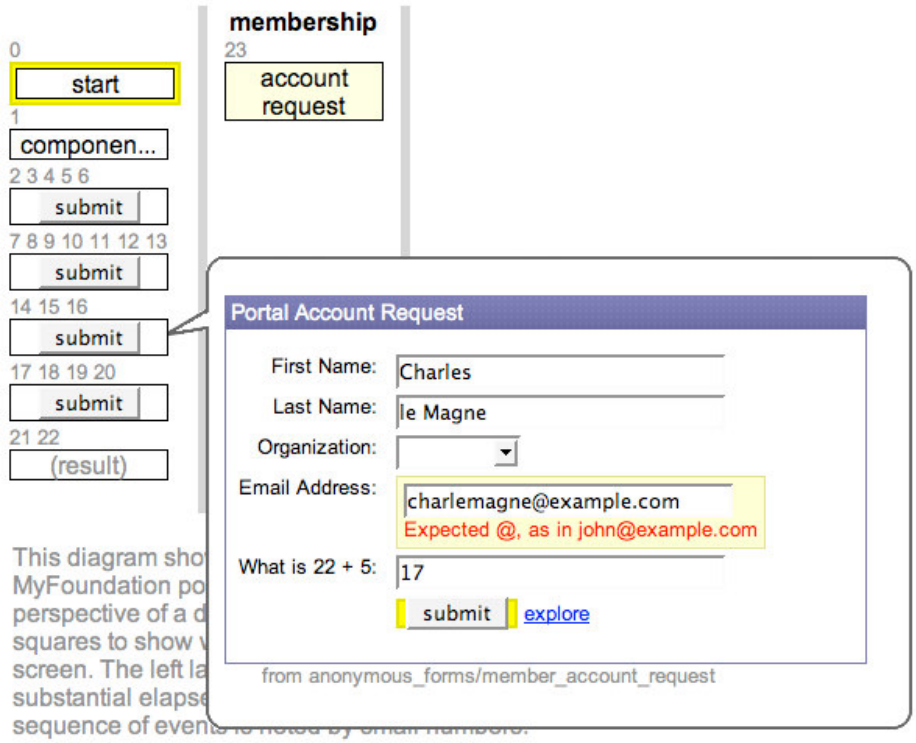
The icons contain a visual summary of the script statement, such as a button or an href link for a press() statement, a yellow title for an email, or a purple note. Errors are, of course, red. Application objects can provide an *effective label* if the default label is not sufficiently descriptive. The entire HTML result is available via a fly-out (implemented with Javascript).

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007



{a fly-out shows exact screen image}

The harness.php functions know the form and format of the standardized HTML that our PHP application objects return, enabling some simple pattern-based HTML augmentation for highlighting the modified form fields and/or pressed action buttons.



{a fly-out includes highlighting of active areas}

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007

Conclusion

Our system has been in continuous use for over six months now - not enough time to declare it "the future of all things good and wonderful", but definitely enough time for us to do make some initial conclusions.

The Eclipse Foundation staff has been happy with the results. Our team (the three of us) is judged helpful, responsive, and productive. Other staff members want to add their processes (long running transactions) to the portal, either with our assistance or on their own. The Foundation staff has shown a definite preference for automation using this framework over the other frameworks and tools available on our servers.

Perhaps the highest compliment we received was when one of our customers (SC) said that she easily understood how everything worked because it "uses her words" in the interface panels and documentation.

We believe that there have been two major reasons for our success to date: First, we've focused on supporting conversations: conversations between developers, conversations between developers and sysadmins; conversations between developers and users; and even conversations between users and other users.

Second, we've worked at, and visualized, the right level of abstraction. Our abstractions are simple enough to understand because we don't talk in computer terms such as "database updates", "ajax form posts", or "user1"; instead we use the user's vocabulary, show the user's interface panels, and use the names of real people and roles in our simulations. At the same time, our abstractions are detailed enough to be believed because we are running the real code in our simulations instead of bubbles and arrows diagrams. We respect the user's expertise without dumbing down our system or forcing them to use our terminology.

References

[Rummler & Brache, 1995] Improving Performance: How to Manage the White Space in the Organization Chart. Jossey-Bass, 1995. ISBN 0787900907.

[Cunningham, Freeman-Benson & Matthias 2007] Swim System Implementation. Download from http://wiki.eclipse.org/MyFoundation_Portal_Implementation

First published at the 25th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2007