Mashed layers and muddled models: debugging mashup applications

M. Cameron Jones
Elizabeth F. Churchill
Internet Experiences Group, Yahoo! Research,
{mcjones, echu}@yahoo-inc.com

Les Nelson
PARC
lesnelson@acm.org

Keywords
Mashups, coding, programming, debugging, collaboration, mental model, comprehension, design, faulty diagnosis, internet, Web, Yahoo! Pipes

INTRODUCTION

Debugging is an essential part of computer programming or coding. An article on Wikipedia defines computer programming to be "the process of writing, testing, debugging/troubleshooting, and maintaining the source code of computer programs" (Wikipedia, 2008). Estimates on the amount of time developers spend testing and debugging applications varies widely. Tassey (2002) reports that over half of development costs are spent on identifying and correcting defects, and between 10% and 35% of programming effort is spent debugging and correcting errors. Ko & Myers (2004), citing Tassey's report, claim that program understanding and debugging represent up to 70% of the time required to ship a software application. Beizer (1990) estimated that as much as 90% of all labor resources spent on software development are spent on testing.

Despite the recognized importance of debugging in software development, it is only recently that researchers have studied debugging in the context of programming work done by non-expert programmers. Studies of "end-user programmer" debugging has primarily focused on individual programmers, in desktop application programming environments (e.g., spreadsheets Kissinger et al, 2005), studied in experimental laboratory settings (e.g., Ko & Myers, 2004). However, debugging practice can also be observed on the web in online communities of developers, spanning both end-users and expert programmers; and encompassing varying degrees of collaborative interaction from loosely-connected independent developers using common tools, to more formal, collaborative, tightly-knit open-source development teams. The artifacts being created and debugged by these "user-programmers" (a term we have adopted in this chapter to reflect the broad range of expertise of those engaged in development activities online) are themselves, often accessible and interrogable online

The term "user-programmer" is not one we have seen commonly used in the literature. End-user programmer appears to be the preferred term; however, this places an emphasis solely on those who are typically perceived as consumers of technology, and highlights their role as producers. It is very difficult to characterize any given community of developers on the web as being "end-users", especially if that community is of sufficient size. The reality is that there are users of varying degrees of expertise, each possessing unique sets of skills. User-programmer, as a label, not only includes end-users, but likewise does not exclude programmers with more experience and expertise.

Classic models of debugging

In traditional models of software development, such as the waterfall and spiral models, testing and debugging software comprise discrete stages in the software development process. Alternative methods emphasize integrated and continuous testing and evaluation of software as it is being written; extreme programming, for example, advocates writing tests before the code is written. However, debugging in

general can be treated and understood, much as it is described by Hailpern and Santhanam (2002) as “the process of … analyzing and possibly extending (with debugging statements) the given program that does not meet the specifications in order to find a new program that is close to the original and does satisfy the specifications.”

They go on to articulate the granularity of analysis required as the code being developed changes state from early development through finished product. In the early stages, debugging tends to be at the lowest code levels, such as stepping through individual code units one statement at a time to verify the immediate computational results. In the latter stages, larger portions of the system are being tested, resulting in increased complexity of debugging across many components and layers of the software.

Eisenstadt (1997) reports on 56 anecdotes for bug "war stories", primarily in C/C++ contexts, and codes these according to "why difficult", "how found", and "root cause". The majority of reported worst case bugs were attributed to remotes (in space or time) of the causes and symptoms of bugs and the lack of visibility in the debugging tools to see the bug.

Agans (2003) presents nine guidelines characterizing successful debugging across the range of software development kinds. Issues of particular relevance to working with mashups include:

- **Understand the system**: Know the entire system, including the reliance on external components and the interface complexities associated with invoking those components.
- **Make it fail**: The conditions needed to stimulate the failure are more complex. The remote nature of the invocation makes both the definition of initial conditions and recording of resulting actions more difficult.
- **Quit thinking and look**: While observation of faults is essential to debugging, building in instrumentation is not an option for mashup sources.
- **Divide and conquer**: The boundary of the API defines the granularity of debugging. Once the bug is traced down the 'rabbit hole' of a Web service, other techniques must come into play.

Law (1997) describes the tools and strategies involved in debugging. Issues of particular relevance to mashups are largely constrained by the API boundary:

- Static debugging consisting of reviewing requirements, design, and code is subject to the visibility allowed by each of the source APIs.
- Run-time observation of the program behavior as a whole at run time is a primary tool in debugging mashups.
- Source code manipulation is limited to instrumenting at the API boundary.
- Dynamic debugging at detailed level including breakpoints, single-stepping and hardware traps is not an option for external sources.

MENTAL MODELS

A mental model is an explanation of how something works; a translation of a system or phenomenon into a cognitive representation of how it works. While analogous to the original systems or phenomena they represent, models are often imperfect, yet they allow people to reason about complex systems, and make predictions which can be translated back to the original context (Craik, 1943). There is a long history of research into mental models and technology design, with a more or less strong commitment to the nature of the model itself or how it is used in human reasoning. The strong notion of a mental model is a fairly complete simulation of the workings of a process – for example a simulation of how a piece of code may transform input to produce output. The more often cited and weaker notion of a mental model is a loose ad hoc narrative explanation for how something might work – sufficient only to allow someone to interact with an interactive device, but not necessarily geared towards problem solving or fault finding and repair. (Norman, 1988).

A slightly different take on debugging is the psychology of programming (PPIG) approach which focuses on the ways in which people debug – that is how human problem solving proceeds given the identification of an error to locate and rectify the bug. Many of these studies focus on the level of expertise with regard to programming, to the language being used, to the environment being programmed within and to the level of complexity of the problem. Some work has been done on the impact of domain expertise on the specification and implementation of the program itself, but this is more seldom reported.

Much of the work on mental models and computer programming is fairly old, dating from the late 1970's through to the mid 1990's. In many instances the issues of mental models and training were perhaps offset by the development of better programming environments and debugging tools, and better help manuals. However, for understanding debugging in more ad hoc environments like the web we believe some of these approaches are still of interest.

Brooks (1983) formulated a cognitive model of how programmers understand code. The model is basically a top-down approach with development of an abstract overview of the domain and the purpose for the code by forming, testing, refining and verifying hypotheses. This kind of top-down approach stands in contrast to proponents of a more bottom-up approach like Shneiderman and Mayer (1979), and Pennington (1987). These researchers consider it more likely that programmers group code into what they consider to be logical chunks by reading the code and formulating chunks into a higher level picture of the overall architecture and its subcomponents. Pennington, more than Shneiderman and Mayer focused on the flow of control and the sequence of events from the program. She proposes two models: first the model of the program based on the control-flow abstraction. Then there is a situation model based on the data-flow/functional abstraction of the program. An integrated meta-model was developed by von Mayrhauser and Vans who offer a mode combined approach. Programmers move between different models as appropriate – top down, situation, program and knowledge base. The knowledge base is the programmer's current knowledge that is used to guide the models as well as store new knowledge.

THE CURRENT SITUATION: WEB DEVELOPMENT

The web, specifically the web browser, has been described as a "really hostile software engineering environment" (Douglas Crockford, http://video.yahoo.com/watch/111593/1710507). Developing applications for the web and the modern web browser presents unique challenges that stymie even the most skilled developer. User-programmers must navigate a complex ecosystem of heterogeneous formats, services, tools, protocols, standards, and languages, and somehow scrape together an application which presents a consistent experience across multiple platforms. The pains of web development are perhaps most evident in the context of web mashups, as these applications draw upon multiple sources of information and services, and necessarily must negotiate the incompatibilities arising from the integration of unlike components. Furthermore, there are numerous tools and services targeted at user-programmers which support mashup development.

Thus there are many sources of complexity: at least two to do with sourcing and one to do with production: (1) multiple sources of information and services, each with different protocols for access; (2) multiple standards for (and use of) language and format; (3) many different tools for creating mashups. Although less frequently the case with the "modern" web browser, it is still sometimes the case that the final production, as rendered by the browser itself can also cause problems.

Mashups are largely ad hoc integrations for situational applications. In this they are akin to research prototypes, designed to quickly 'cobble together' enough of a solution from existing resources to demonstrate novelty in capability and/or user experience. From experiences of research integrations, Nelson & Churchill (2006) report the diagnosing of errors through layers of external applications and operating system software. In this instance involving production of printed cards for a tangible interface for controlling Microsoft Powerpoint, the assumptions and limitations of Powerpoint, Microsoft Word, and the various print drivers were very much a factor involved in diagnosing problems in the new application.

Debugging of mashups instriniscally involves the space-time distance and visibility issues due to distributed and black-box dependencies. And also as in the mashup situation, it falls to the end-user to make

the diagnosis and correct the problem through the many models of computation involved. Current web development environments incorporate many layers of abstraction, and depend on many features which are black-boxed, i.e., an opaque abstraction that does not yield to inspection or interrogation. What differentiates web development from other development contexts is that the black boxes and abstractions of the web, are neither uniform nor consistent, and often developers must employ complex articulations in order to align entities which operate at different levels of abstraction, or execute in different layers of the execution stack. For example, in the snippet shown in Figure 1, a PHP array reference (`$item`) is embedded in a line of JavaScript code (`var asin...`), which is in an HTML document, being generated by a PHP script. The PHP script is parsed and interpreted on the server, generating an HTML document as output. The HTML document, which contains dynamically written JavaScript, is passed to the client's browser, where it is executed locally.

Keeping track of when particular pieces of code are executed, and in what context is difficult for professional developers, and even more so for end-users. In Jones and Twidale (2006b), we learned that a major obstacle for novice programmers in building a mashup is maintaining an accurate mental representation of the process model, and tracking when and where each code segment is executed. Expert developers will recognize the ugliness of the code in Figure 1, and suggest introducing additional layers of abstraction (e.g., using a Model-View-Controller pattern) as a means of managing the complexity. However, the addition of new layers of abstraction also adds more layers which must be learned, understood, and examined upon failure.

```
<?php
...
?>
<html><head>
<script type="text/javascript">
var asin = "<?php echo $item['Asin']; ?>";
...
</script>
<?php
...
?>
```

Figure 1. Messy Mashup Code. The PHP code blocks have been highlighted.

Another factor which confounds mashup development is the reliability of the underlying services and infrastructure on which an application is built. For example, one mashup application developed two years ago by one of the authors recently stopped working because the mapping service it used changed its API. Another mashup stopped working because the organization providing the service changed its name, and moved its servers to new domains, breaking the connection. It is seemingly a contradiction to have a "robust mashup application", as the developer has little to no control over the remote services and information sources being used. It is like building a house on sand, the ground is constantly shifting, making it difficult to make any long-term progress.

Numerous user-level tools and programming environments have been developed which attempt to facilitate mashup development: Intel Mash Maker, Yahoo! Pipes, and Microsoft Popfly are some of the most widely known of these. We have been studying Yahoo! Pipes and the community of users who engage in discussions on the Yahoo! Pipes developer forums.

YAHOO! PIPES

Yahoo! Pipes is a web-based visual programming language for constructing data mashups. Yahoo! Pipes was originally developed as a tool to make extracting, aggregating, and republishing data from across the web easier. Since its launch in February 2007, over 90,000 developers have created individual pipes on the Yahoo! Pipes platform, and pipes are executed over 5,000,000 times each day. Figure 2 shows the Yahoo! Pipes editing environment; it consists of four main regions: a navigational bar across the top, the toolbox on the left, the work canvas in the center, and a debug-output panel at the bottom. The toolbox contains modules, the building blocks of the Yahoo! Pipes visual language.

Figure 2. Yahoo! Pipes Visual Mashup Editor

Yahoo! Pipes' namesake is the Unix command-line pipe operator, which allows a user to string together a series of commands, where the output of one command is passed as input to the next. In the graphical language of Yahoo! Pipes, modules (operators) are laid out on a design canvas. Modules may have zero or more input ports, and all have at least one output port; additionally, modules may have parameters which can be set by the programmer, or themselves wired into the output of other modules so that the value of the parameter is dependent upon a runtime value specified elsewhere. The input and output ports are wired together, representing the flow of data through the application. Selecting an output port, highlights all the compatible input ports to which the output may be connected.

There are a number of data types within Yahoo! Pipes which determine what inputs and outputs are compatible. In the most general terms, there are simple scalar data *values*, and *items*, which are sets of data objects (e.g., items in an RSS feed, or nodes in an XML document). Values have types, including: text, urls, locations, numbers, dates, and times.

In Yahoo! Pipes, data flows from the initial module(s), where user-data are input, or external data are retrieved, through subsequent modules in the pattern and order dictated by the wiring diagram. All applications in Yahoo! Pipes have a single output module, which is wired to the end of the execution sequence, and collects the final data stream for distribution via RSS, JSON (JavaScript Object Notation), or a variety of other formats. Drawing on the Unix command-line metaphor, the output module is akin to "standard out" or the user terminal.

Unlike the Unix command-line pipe, Yahoo! Pipes allows users to define complex branching, and looping structures, have multiple sources and execution paths executing in parallel, and in general, create programs of arbitrary complexity. There is no direct method for writing recursive functions, and Yahoo! Pipes does not allow for cycles in the program structure (i.e., where the output of a module is fed back to the input of a module further 'upstream'). This enforces a more-or-less linear execution flow to the applications which is bounded by the amount of data being processed.

SOCIAL COLLABORATIVE DEBUGGING

The web, while presenting many unique challenges for development, does also offer some potential solutions in the form of large communities of programmers willing to engage in conversations about programming problems. These conversations not only service the original participants, but also become searchable archives which continue to support subsequent developers in programming and debugging. The conversations also provide rich evidence of users articulating their problems and translating their understanding across literacies, and mental and computational models.

Traditional models of debugging and empirical studies of debugging tend to focus on single coders, solving contrived single "bugs", in closed stable worlds. The web and mashups therefore do not really map well to many of these models of how debugging takes place in practice, although the recommendations for systematic investigation, hold.

A somewhat different programming situation that also requires the kind of development of a mental model of the code is that of understanding legacy code. Here, programmers need to understand the original programmer's thoughts, decisions, and the rationale behind the design of the legacy code. Burgos et al. (2007) studied programmers as they wrestled with the development of models of legacy code in organizations and came up with six main methods that programmers use: software documentation, skills databases (to find experts), knowledge databases (where project data like design documents and traceability matrices are stored and made available), person-to-person communication, source code (searching for key identifiers) and reverse engineering tools.

Obviously, source code is a primary resource in debugging an application. There are numerous tools and resources on the web for sharing source code, and source code snippets; many of these tools have been designed explicitly to support sharing code for the purposes of debugging. One such tool, Pastebin (both an open-source software toolkit, and a web service pastebin.com), describes itself as a "collaborative debugging tool". Pastebin allows user-programmers to paste source code (or any other text) into a web form; the source code is then persisted for a user-specified time period on a server and given a unique URL.

The URL can then be distributed to other people who can view the source code which was originally pasted and shared. In addition, other user-programmers can post corrections or amendments to the original snippet.

Services like Pastebin originated out of a need to establish a common context or focus around a particular segment or snippet of code. This is very hard to do in instant messaging, where messages are limited in length; in public IRC channels pasted code would flood a channel with streams of text making it difficult, if not impossible, to have a conversation; and it is also difficult to manage in discussion forums as the code gets entangled with the discussion thread. Additionally, making comments or revisions to the code in these contexts requires repeatedly copying and pasting the code into the conversation, making it difficult to read and effectively compare revisions.

While Pastebin allows any arbitrary text to be posted, it is typically used for sharing source code snippets, console logs, and error logs. Other similar code-sharing oriented tools include services like gist.github.com, and pastie.org; and there are numerous services for sharing snippets which are not directly targeted at source-code, including Thumbtack and Live Clipboard both from Microsoft Live Labs, amd Google Notebook. These services satisfy varying degrees of collaborative interaction, ranging from supporting awareness to facilitating collaborative coding. Jones and Twidale (2006) showed that shared copied text among collaborators can be used to infer activity and attention, which helps maintain awareness in collaborative environments. Some of these tools support annotation, allowing other people to comment on, or suggest changes to the code; other tools support full collaborative editing, complete with access control and revision management (e.g., Gist is a snippet sharing service backed by a git version control repository). The Yahoo! Pipes platform supports similar functionality to Pastebin, in that each pipe application is individually addressed by a unique ID and URL. Users may publish their pipes in the public directory, where they can be searched, browsed, and viewed by anyone. However, Yahoo! Pipes has a very open security model, allowing any user to view and run any pipe, so long as they know the URL, even if it is not published in the directory. This design was intentional, as the Yahoo! Pipes developers wanted to foster the kind of learning-by-example which Netscape's "View Source" feature facilitated for HTML. Thus, every pipe application which is created has a "View Source" button attached to it, allowing users to inspect how a pipe works. This allows users not only to share links to their in-progress, and unpublished pipes, but view and modify each other's pipes; allowing users to collaboratively debug problems.

When a pipe application is viewed by a user who is not the owner of the pipe, a local copy is made in the user's browser, and any changes which are made by the user are then saved to a new copy on the server, preserving the original pipe. Additionally, each pipe has a "clone" button, which allows a user to make a copy of an existing pipe (either their own or someone else's).

In addition to entire pipes being copiable and modifiable, pipes can be embedded within one another as a "sub-pipe". This allows developers to create and share reusable components, and generate intermediate levels of abstraction in their applications. An embedded sub-pipe is represented as a module in the Yahoo! Pipes interface, which can be wired to other modules, or other sub-pipes. Users can drill-down into embedded sub-pipes, to inspect and modify the included functionality.

Yahoo! Pipes provides discussion forums for users to talk, ask questions, interact with each other and the Pipes development team, and give and receive help. The Yahoo! Pipes discussion boards (http://discuss.pipes.yahoo.com/) have been active since Pipes was launched in February, 2007. The forums have been a significant source of information on programming in Pipes, as there is not much documentation for the language, merely some tutorials and annotations. We have studied monthly snapshots of the Pipes discussion forums sampled from the past two years, the most recent snapshot ending 1 December 2008.

The discussions in the Pipes forums are divided into three areas: Developer Help, General Discussion, and a section for showing off Pipes applications. Table 1 provides some general statistics about the activity of the forums at the most recent snapshot (December 2008). Most of the activity on the Pipes discussion forums is question-answer-type interactions in the Developer Help forum. The majority of posts receive replies of some kind. However, just over one fourth (25.9%) of posts never receive a response, and that proportion falls to 20% when we consider just the help forum, where most of the debugging conversations occur.

Table 1. Statistics of the Pipes developer forums from December 2008.

Pipes Forums Activity Data	Developer Help	General Discussion	Show Off Your Pipe
Number of Threads	**1731**	**576**	**241**
Number of posts without replies	**347**	**165**	**149**
Avg. Thread Length	**3.65**	**2.89**	**1.07**
Std. dev. of Thread Length	**4.22**	**3.48**	**1.53**
Number of participants	**1523**	**638**	**236**
Avg. num. participants per thread	**2.34**	**2.14**	**1.34**
Std. dev. of num participants	**2.13**	**2.06**	**0.69**

Most user-programmers do not participate in the discussion forums. Of the over 90,000 user-programmers who have authored a pipe application, only 2,081 have posted a message to one of the forums. Nearly all (1,725) of them participate in the forums for a month or less. Figure 3 shows the distribution of participant engagement in the Pipes discussion forums. There is a high degree of transiency in the community as people come, engage for short periods of time, and leave.

Figure 3. The distribution of how long user-programmers have participated in discussions in the Yahoo! Pipes forums. Most user-programmers only participate for a short time.

The community and social structure which emerges from the interactions in the Pipes forums is characterized by several strong hubs, individuals who actively seek to engage new-comers and answer questions. These hubs include members of the Pipes development team, as well as peer user-programmers from the community. There are, however, relatively few hubs and the majority of participants in the discussions are connected to very few other individuals: 16% (336) participants are not connected to anyone else; 63% (1311) participants are connected to five or fewer other individuals; 12 participants are connected to 100 or more others in the network, with one participant having over 1,000 connections. The social network of interaction, measured by mutual participation in a thread on the forum, is depicted in Figure 4. In this network, nodes represent participants, and edges represent a social tie between two participants if they have participated in a common thread; the relationship is unweighted (i.e., binary).

Figure 4. The social network of the Yahoo! Pipes discussion forums. There is a clear separation between a connected core and a periphery of isolated individuals and small cliques.

The social network depicted in Figure 4 highlights several features of the Pipes community. While the elliptical shape and egg-yolk appearance of this network visualization is primarily a visual artifact of the algorithm used to render it, there are none-the-less meaningful structures in the graph which reflect the structure and dynamics of the community. One of the most striking visual structures is the clear separation between a connected core, and a periphery of isolated individuals and cliques. Within the core of the interconnected individuals there are observable valences of engagement (i.e., more connected individuals are closer to the center), and several social hubs can be readily seen. The fan-shape arrangement in the lower-right quadrant of the center is anchored by an individual with over 1,000 connections. This participant is engaging in a large amount of question-answering, acting as the sole respondent many new-comers who tend to leave the forums after getting the answer they came for.

In the periphery of the network are individuals who have no social ties to other members of the community, and isolated cliques, or small groups of dyads and triads who interact with each other exclusively. The emergence of this structure within the community is intriguing. Why do some questions get answered and

others not? How do people move from the periphery to the core? How do user-programmers engage with communities over time? Figure 5 shows the growth of the community in monthly intervals over the 22 months during which the forums have been active, broken down into the number of disconnected individuals, connected individuals, and individuals who were newly connected in the time-period sampled (both new to the community, and existing members who were previously disconnected).

Figure 5. The growth of the Pipes discussion forums community from 07 February 2007 to 01 December 2008, illustrating the relative growth in the numbers of connected and disconnected individuals.

The growth depicted in Figure 5 shows a relatively linear growth in the size of the network as a whole, and both subsets of connected and disconnected individuals tracks a similar trajectory of linear growth. This could be interpreted as evidence that the community is adequately responding to new posts and questions asked by new members, otherwise we would expect to see the size of the disconnected region growing faster than that of the connected region. In order to better illustrate the detailed dynamics and interactions around debugging in the Pipes community, we present several examples and snippets of conversations, selected from the forums. We focus on conversations involving more than one participant which illuminate the interactions developers have around software debugging and collaborative problem solving.

Worked examples of social debugging from Yahoo Pipes

Tools like Yahoo! Pipes layer on additional abstractions and metaphors in the web development context which can make building mashups faster and easier. However, these additional layers introduce new black boxes into the web development ecosystem which can interfere with user's prior understanding of how things work, and like the findings of Churchill and Nelson (2002), can complicate debugging activities by obscuring the sources of errors.

Consider the following snippet of a discussion taken from the Yahoo! Pipes developer forums. In the following conversation, a user describes the problem he is having viewing the output of a Pipe application. He initially assumes the problem he is experiencing has to do with the application he composed in Yahoo! Pipes.

Paul
I have created an images-only RSS feed that basically grabs image such as these: http://www.vrindavandarshan.com/yr2008/a...
The problem is the images are not showing up in-line as "content", rather you have to click through to actually see them. This makes the RSS feed rather useless for displaying inside other modules or devices. Could someone look at my pipe and let me know what I am doing wrong? All I want is for the output of the RSS feed to simply contain the 3 images of the day. Here's the pipe as far as I have gotten it: http://pipes.yahoo.com/pipes/pipe.info?_...

Roger
When I looked at your RSS feed - http://pipes.yahoo.com/pipes/pipe.run?_i... - the images were displayed. Are you still not seeing the images? If so you will need to supply more details about how you are viewing the RSS feed. It may be that you are experiencing caching issues that should not last more than about 30 minutes.

Paul
I'm trying to use Google Reader, which does not seem to show a "preview" image in-line. I have to click through to each image to see it. Maybe this is just a google reader thing, but I have other feeds (eg, Dilbert web comic) that shows the image right on the page.

Roger
When using "Expanded view" in Google Reader both the Dilbert feed and you feed show images without

any need for clicking. At least, that's what I'm seeing. I can't see any "preview" images for either feed in either List or Expanded view.

Paul
This is very strange, because the feed will not work for me unless I manually load the images by typing the image URL directly into the address bar first. After I do this, the image seems to 'preload' and then the RSS feed works fine. If I don't do this I get a 403 forbidden error from the site. Is there some standard template that can be used for creating an RSS feed that is simply a group of images?

Roger
I think we are in the realm of browser/operating system/security settings differences.
All I can say is that the feed works for me in Google Reader using FF3 and W2K.
You may be able to find more help from the Google Reader Help group.
http://groups.google.com/group/Google-Re..

*All user names have been replaced with pseudonyms.

Figure 6. An exchange between two Pipes developers where they are attempting to localize the source of a bug, peeling off the layers of the web programming stack.

In the exchange between Paul and Roger, we can see the progressive peeling away of layers of abstraction and execution. Paul begins with the assumption that there is a problem in his pipe application, that he has caused an error. Roger replies stating that the pipe appears to work for him, and does not exhibit the problematic behavior Paul is reporting; he offers the suggestion that the problem may be in the caching behavior of the Yahoo! Pipes platform. Paul responds that he thinks it might be a problem of the Google Reader RSS viewer application, and not a problem with his pipe or the Yahoo! Pipes cache. After several more exchanges, Roger asserts that the problem likely is being caused by an incompatible browser or operating system setting.

In this example, we can see that the potential sources of the errors are far more numerous than typically considered in debugging. Typically, when a program does not work as expected, the programmer assumes there is a problem in his/her code, trusting that the underlying compiler/interpreter, operating system, networking stack, etc. are working properly. Rarely would we expect an error in program execution to be caused by a bug in the underlying operating system, for example. However, in Yahoo! Pipes, the underlying infrastructure for interpreting and executing the pipe application may itself have bugs; or the browser or other application in which the pipe is being executed, or the output is being rendered may be incompatible with certain aspects of the Yahoo! Pipes system or data formatting; or there may be a problem with improperly formatted data being fed into the pipe; or some other problem further upstream in one of the data sources. Many of these problems are outside the user's control, making them nearly impossible to resolve.

Over the course of the discussion, the source of the error shifts, touching on nearly every layer of the programming stack, and in the end, the user is no closer to solving the problem than he was at the beginning. In highly abstracted programming contexts like that of web mashup programming, the number of possible sources which must be examined increases, which presents challenges for user-programmers in identifying, characterizing, and localizing bugs in their applications. While this problem may be particularly salient in Yahoo! Pipes context, given the additional layers introduced by the visual language, we believe these problems to be inherent in mashup programming in general. Web mashups are necessarily embedded in a web of interdependent services, platforms, and data objects, many of which are not as robust or verified as modern compilers, or the underlying operating system stack. While web mashups are often discussed in the context of the "web as operating system", the reality is that the web is not as stable or robust as a standard desktop operating system. It is often that case that services have bugs or fail, network connections are not reliable, and data are not properly formatted (often because the standards are underspecified). Jones, Churchill, and Twidale (2008) framed these challenges within the cognitive dimensions framework. They argue that the existing cognitive dimensions do not account for the additional complexity and challenges imposed by the open, heterogeneous nature of the mashup ecosystem, and point

to the affordances development tools, like Pipes, have for sharing and collaborative debugging as a possible mechanism for cognitive offloading, and effective resolution to complex problems.

Other bugs stem exclusively from problems with the visual language of Pipes itself, and have little to do with the mashups more generally. Figure 7 contains a snippet of conversation where several user-programmers are reporting problems with the drag-and-drop interaction in the Yahoo! Pipes interface. These user-programmers are reporting problems dropping modules into a loop operator module.

Pete
Am I just missing something very basic here but how do I drop a source module onto the For Each: Replace operator. No matter what I try, the boxes will not merge like in the examples.
So to be clear I am trying to drop a Flickr source that I am trying to drop (merge) with a For Each: replace operator.
I am using firefox 2.0

Freddy
I want to chime in. I'm having problems getting the for each operators to work. After I create a for each: replace I can not drop anything onto it. I try flickr or fetch and they simply lie over it.
This is both FF 1.5.0.9 and Safari 2.0.4 on mac (intel). Any help with this would be greatly appreciated.

Seven additional posts in which developers document different browser and operating system configurations have been omitted

Harry†
The browsers mentioned, in particular FF and IE7, should have no problems with this. We'll look into what's causing it. It doesn't seem to always crop up. I know it's not ideal but sometimes even when it doesn't work for an existing Pipe, a new one will still work.
To help us figure it out, feel free to post Pipe URLs where you're seeing this issue or talk about the steps leading up to it. Thanks!

Freddy
I've tried on ff on a few platforms, no go for me... I've created new ones, restarted the browser, restarted the machine. :)
It doesn't work in any pipe I use, but here's one:
http://pipes.yahoo.com/pipes/knwB40642xG...

Two posts about style-sheet errors have been omitted.

Don†
We're still looking into this - but having trouble reproducing it (Im sure you are all seeing it tho).
One thing that occurred to me that isn't as obvious as it should be, is that you can't drag modules that are on the canvas INTO the foreach - you can only drag them from the "mi pipes" or "sources" toolbox/library on the left (or the search results from the top right). Is that one of the problems here?

Freddy
Oops. Yes that was the problem. I didn't even realize that you could drag off the left nav - I was just clicking them and having the modules show up on the canvas.

*** All user names have been replaced with pseudonyms**
†denotes a Yahoo! employee working on the Pipes platform.

Figure 7. Excerpts from a thread relating to some confusion about the drag-and-drop functionality in the Yahoo! Pipes visual editor.

In the above example, but the “bug” which is being fixed is not necessarily technical, but rather is a “bug” in the mental models being formulated. Their conversational interactions illustrate not only the bugs in Freddy and Pete’s models of how Pipes works, but also in Harry and Don’s (both who know how Pipes

works because they built it) models of what the problem is and their understanding of the models which user-programmers have of Pipes. There emerges a sense that the problem is perhaps in the browser and operating system configuration, that the user-programmers experiencing the problem are using an unsupported or untested system. There was even a section of the conversation pertaining to stylesheet errors as a possible cause of the problem.

The exploration of the numerous possible sources and their entailments highlights how complex and interdependent the ecosystem is. Even when powerful, robust tools for authoring exist which seek to facilitate development by substituting relatively homogenous abstractions for the mixed, ad hoc, and or conflicting ones the user-programmer would have to address otherwise, the user-programmer is only one veneer away from the messy and ugly issues, raising questions about how much can be taken for granted, and how much knowledge is needed in order to effectively author applications in environments like Yahoo! Pipes. Similar to how a Windows programmer works with abstractions like the file-system without much need to know the internals of how it works; can the web reach a point where user-programmers do not need intimate knowledge of all the component parts in order to effectively author applications?

CONCLUSIONS

In this chapter we have described the social and collaborative practices of debugging among user-programmers in the Yahoo! Pipes community. The highly abstract and layered architectures of web programming present numerous challenges to user-programmers. They can obscure the source of problems, complicating the challenge of localizing the source of a bug; and the abstractions they add can not only introduce new bugs, but also conflict with established understandings and lead to wasted effort chasing red-herrings in the debugging process.

We close this chapter with several questions about the future of end-user programming for the world-wide web. Do authoring tools like Yahoo! Pipes actually support true “end-users”? How much knowledge of the underlying infrastructure, protocols, and technologies is needed to get started? To be proficient or expert? Do mashup authoring tools allow users to express themselves easily and intuitively? How exportable is knowledge from one environment to another? How do we encourage deep learning in abstracted environments? Do these tools impose models which conflict with the underlying data or technology? Or conflict with users’ prior knowledge or understanding?

REFERENCES

1.Agans, D., “Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems”, New York, AMACOM, 2003.

Burgos, C.L., Ryan, J.J.C.H and Murphee, E.L. Through a mirror darkly: How programmers understand legacy code. Information Knowledge Systems Management, 6 (2007), 215-234, IOS Press.

Brooks, R. Toward a Theory of Comprehension of Computer Programs. International Journal of Man Machine Studies. 18(6), 1983, 542-554

Churchill, E.F., Nelson, L.D., Tangibly Simple, Architecturally Complex: Evaluating a Tangible Presentation Aid, CHI ‘02 Extended Abstracts on Human Factors in Computing, ACM Press, 2002

Craik, K. J. W. The Nature of Explanation. Cambridge University Press, 1943.

Curtis, W. 1988. Five paradigms in the psychology of programming. In M. Helander (Ed) Handbook of Human Computer Interaction. Amsterdam, North-Holland.

Eisenstadt, M. (1997). My hairiest bug war stories. Communications of the ACM, 40(4), 30-37.

Fleischmann, K. R.; Wallace, W. A. (2005). A Convenant with Transparency: Opening the Black Box of Models. CACM 48(5).

Gilmore, D.J. Models of debugging. Acta Psychologica 78 (1991) 151-172

2.Hailpern, B., Santhanam, P., Software debugging, testing, and verification. IBM Systems Journal 41(1): 4-12, 2002.

Jones, M. Cameron (2007). Copy-Paste Programming: An exploratory analysis of software clones in Open-Source. Classification Society of North America (CSNA) 2007.

Jones, M. Cameron; Elizabeth F. Churchill; Michael B. Twidale (2008). Mashing up Visual Languages and Web Mashups. VL/HCC 2008.
Jones, M. Cameron; Elizabeth F. Churchill; Michael B. Twidale (2008). Tinkering, Tailoring, & Mashing: The Social and Collaborative Practices of the Read- Write Web. CSCW Workshop. San Diego, California.
Jones, M. Cameron; Michael B. Twidale (2006a). Snippets of Awareness: Syndicated Copy Histories. CSCW 06 Poster. Banff, Alberta, Canada.
Jones, M. Cameron; Michael B. Twidale (2006b). Web Mashups and CSCW: Opportunities and Issues. CSCW 06 Workshop. Banff, Alberta, Canada.
Katz, I. and Anderson, J.R. Debugging: An analysis of bug location strategies. Human Computer Interaction, 3, 351-400.
Kissinger. C. et al. (2006). Supporting End-User Debugging: What Do Users Want to Know? AVI06.
3.Ko, A. J., Myers, B. A. (2004) Designing the whyline: a debugging interface for asking questions about program behavior. ACM SIGCHI 2004, 151-158.
4.Law, R., An overview of debugging tools, ACM SIGSOFT Software Engineering Notes, v.22 n.2, p.43-47, 1997.
Nelson, L. D.; Churchill, E. F., (2006). Repurposing: Techniques for reuse and integration of interactive systems, Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration, pp. 490-495, 2006.
Nelson, L. D.; Smetters, D., Churchill, E. F., Keyholes: selective sharing in close collaboration, CHI '08 extended abstracts on Human factors in computing systems, 2008.
Norman, D. (1988). Psychology of Everday Things. Basic Books.
Pennington, N. Stimulus Structures and Mental Representation in Expert Comprehension of Computer Programs. Cognitive Psychology 19 (3), 1987, 295-341.
Scaffidi, C., Shaw, M., Myers, B. (2005). Estimating the Numbers of End Users and End User Programmers. VL/HCC 2005.
Shneiderman, B. and Mayer, R. Syntacti/semantic interactions in programmer behavior: A model and experimental results. International Journal of Parallel Programming, 8 (3), 1979. 219-238.
Tassey, G. (2002). NIST: The Economic Impacts of Inadequate Infrastructure for Software Testing.
5.Wikipedia, Computer Programming. http://en.wikipedia.org/wiki/Computer_programming. Accessed February 19th 2009.