Identification, Impact, and Refactoring of Smells in Pipe-Like Web Mashups

Kathryn T. Stolee, *Member*, *IEEE*, and Sebastian Elbaum, *Member*, *IEEE*

Abstract—With the emergence of tools to support visual mashup creation, tens of thousands of users have started to access, manipulate, and compose data from web sources. We have observed, however, that mashups created by these users tend to suffer from deficiencies that propagate as mashups are reused, which happens frequently. To address these deficiencies, we would like to bring some of the benefits of software engineering techniques to the end users creating these programs. In this work, we focus on identifying code smells indicative of the deficiencies we observed in web mashups programmed in the popular Yahoo! Pipes environment. Through an empirical study, we explore the impact of those smells on the preferences of 61 users, and observe that a significant majority of users prefer mashups without smells. We then introduce refactorings targeting those smells. These refactorings reduce the complexity of the mashup programs, increase their abstraction, update broken data sources and dated components, and standardize their structures to fit the community development patterns. Our assessment of a sample of over 8,000 mashups shows that smells are present in 81 percent of them and that the proposed refactorings can reduce the number of smelly mashups to 16 percent, illustrating the potential of refactoring to support the thousands of end-users programming mashups. Further, we explore how the smells and refactorings can apply to other end-user programming domains to show the generalizability of our approach.

Index Terms—End-user software engineering, end-user programming, web mashups, refactoring, code smells, empirical studies

1 Introduction

Mashups are programs that manipulate existing data sources to create a new piece of data or service that can be plugged into a webpage or integrated into an RSS feed aggregator. One common type of mashup, for example, consists of obtaining data from some feeds (e.g., house sales, vote records, bike trails), joining those data sets, filtering them according to a criteria, and plotting them on a map published at a site [2]. The development environments that support mashup languages often contain built-in functionality to quickly access and manipulate data, abstracting away much of the implementation complexity, and allowing the users to focus on the high-level behavior of the mashup.

One popular mashup language and environment is Yahoo! Pipes [3]. Over 90,000 end users have created mashups in this visual mashup development environment since 2007, and over 5 million mashups are executed on Yahoo's servers daily [4]. One example is shown in Fig. 1a. To program these mashups, users drag and drop predefined modules (the boxes in Fig. 1a) onto the canvas, connect the modules via wires (the lines between the boxes), and parameterize the modules by setting their field values (the text boxes and drop-down boxes within the modules). The modules perform various predefined functions, such as retrieving data from a web source (fetch) or selecting a subset of the retrieved data (filter), and act as interfaces to an API. The field values modify the behavior of each module by specifying, for example, the websites from which to fetch the data or the expressions to specify a filter criterion; these serve to parameterize the API call. The wires that connect the modules define the mashup data and control flow. Similar high-level, visual, compositional programming languages, and representations, that we group under the umbrella of *pipe-like mashups*, can be seen across several mashup environments (e.g., Apatar [5], DERI Pipes [6], Feed Rinse [7], IBM Mashup Center [8]).

In spite of the increasing power and popularity of mashup environments, we have observed that mashup programs tend to suffer from common deficiencies, such as being unnecessarily complex, using inappropriate or dated modules or sources of data, assembling nonstandard patterns, and duplicating values and functionality. In our study (Section 7), we studied 8,051 mashups from the Yahoo! Pipes repository, and these programs were littered with code smells. Approximately 23 percent had redundant modules, 32 percent had the same string hard-coded in multiple places, and 14 percent used sources of data that were not working as specified. In total, 81 percent of the pipes had at least one type of deficiency. Using our proposed refactorings (Section 6), the deficiencies can be completely removed in 80 percent of those deficient pipes. A study to assess the impact of smells on end-user programmers reveals that the refactored pipes are generally preferred and that smelly pipes are harder to understand and maintain (Section 5).

The presence of such deficiencies in mashup applications is not necessarily surprising, but it becomes more concerning in light of the fact that 66 percent of the pipes we studied had been cloned for reuse an average of 17 times,

- *K.T. Stolee is with the Department of Computer Science and the Department of Electrical and Computer Engineering, Iowa State University, 209 Atanasoff Hall, Ames, IA 50011-1041. E-mail: kstolee@iastate.edu.*
- *S. Elbaum is with the Department of Computer Science, University of Nebraska-Lincoln, 256 Avery Hall, Lincoln, NE 68588. E-mail: elbaum@cse.unl.edu.*

Manuscript received 22 Oct. 2012; revised 1 July 2013; accepted 25 Aug. 2013; published online 5 Sept. 2013.
Recommended for acceptance by H. Gall.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-10-0301.
Digital Object Identifier no. 10.1109/TSE.2013.42.

(a) Example Pipe in Editor

(b) Example Pipe Execution

Fig. 1. Introduction to the Yahoo! Pipes environment with two views, *edit* and *information*.

even though more than three-fourths of them contain at least one deficiency. The deficiencies that end-user programmers of mashups encounter have similarities with those found by professional programmers and are often referred to as code smells—indications that something may be wrong with a section of code. Software engineers have at their disposal techniques and tools to help maintain their code and address such smells by performing semantic preserving transformations on their programs to remove smells, a process called refactoring [9], [10], [11].

Assisting end users with designing and creating mashups is a well-studied research area, with the goal of making mashups easier and faster to create [12], [13], [14], [15], [16], [17]. Yet, research targeting maintenance is just starting to emerge [18]. Through this work, we investigate how to bring the benefits of software engineering maintenance techniques to end-users programming mashups, which seems reasonable given that 46 percent of the mashups we studied were modified after being created, and 26 percent were modified after being added to the public repository, providing evidence of maintenance activities. Adapting software engineering methodologies to help end-users program more dependably and maintainably in nontraditional language paradigms is not new in the software engineering research literature (e.g., [19], [20], [21]). In the context of professional programmers, refactoring has received considerable attention [22], but the focus on refactoring for end-user programming languages is just beginning for web mashups in our previous work [1] and also for spreadsheets [23], [24].

In this work, we focus on mashup maintenance and understanding through automated smell identification and refactoring. The mashup domain introduces new smells and is amenable to novel refactorings. In particular, through this work, we explore smells and refactoring that: 1) leverage the pipe-like mashup language semantics to simplify a pipe structure, 2) target mashups' intrinsic reliance on external and uncontrolled services and data sources that may change without notice, and 3) utilize the public repositories of mashups to standardize and promote understanding across the community. In addition, this work is the first to explore the impact of code smells on end-user programmers. The contributions of this work are:

- Identification and definition of the most prevalent smells in 8,051 mashups.
- Empirical evaluation of the impact of code smells on 61 users' mashup preferences and understanding.
- Design of domain-specific transformations to refactor smelly pipe-like mashups, and tailoring and assessment of those transformations to the Yahoo! Pipes environment.
- Discussion on extensions of refactoring to other end-user programming domains and languages.
- Availability online of the data and infrastructure used for the study.

This paper extends a previous effort [1] along several dimensions. First, we present in detail a study of 61 users over 18 tasks to reveal the impact of smells on user preferences (Section 5). Second, we have added a discussion of how refactoring could benefit other end-user domains and outlined avenues for future work (Section 10). Third, as the target of this work is an end-user programming language and refactoring was originally developed for professional languages, we include the preferences of end users and degreed users in our study but find no significant differences between the groups (*RQ1* in Section 5). Fourth, examples have been added throughout the smell and refactoring definitions to make them more approachable to the reader (Sections 4 and 6). In addition, Sections 7, 8, and 9 have been extended and completed to provide more details on related work, our study infrastructure, and the threats to validity across all studies presented.

The rest of the paper is organized as follows: A motivating example is discussed in Section 2, followed by the definitions (Section 3) used to define smells (Section 4). A study to evaluate the impact of code smells from a user perspective is presented in Section 5, followed by the definition of refactorings that can remove the smells (Section 6). Next, a study to measure the effectiveness of the proposed refactoring in removing smells is presented in Section 7. The threats to validity are presented in Section 8. Section 9 presents related work in mashups, refactoring, and graph transformations. A discussion of the applicability of smell detection and refactoring to other end-user language domains is in Section 10, and Section 11 summarizes the findings.

2 Motivation

In this section, we present an example to illustrate what a pipe-like mashup is, the Yahoo! Pipes mashup language and environment, the potential smells in such mashups, how refactorings can remove those smells, and how and when users prefer the proposed refactorings.

2.1 Yahoo! Pipes Environment

In the Yahoo! Pipes environment, mashups are programs executed on Yahoo's servers. For any mashup, there are two views through which the developer can access the mashup, the *edit* view and the *information* view; both views are accessed through the web browser.

To program a pipe, the user loads the Pipes Editor, which is the *edit* view of the pipe. Fig. 1a shows a screen shot of a browser displaying the Pipes Editor, the development environment housing the visual and compositional language Yahoo! Pipes. Through this environment users can program mashups by dragging and dropping existing components from the module library on the left onto the canvas on the right. Each box in a pipe is a module, whose behavior is defined by the Yahoo! Pipes environment, and is connected to other modules via wires. Each module has a name (e.g., *truncate*, *filter*), and most modules contain fields that can hold hard-coded values or receive values via wire. Modules can be configured by setting field values (e.g., a URL). The structures of these pipe-like mashups are best understood from top (inputs) to bottom (output). Starting from the data sources that serve as inputs, a pipe-like mashup combines and manipulates those inputs to create exactly one output. In Fig. 1a, the *fetch feed* module accesses an RSS feed, the *filter* module blocks items with the word "tennis" in the description, the *truncate* module retains the first five items in the RSS feed, and the *pipe output* serves as the output of the pipe.

As of June 2013, the Yahoo! Pipes language has 46 supported modules and four deprecated modules. Among the pipes we scraped from the repository for our study (Section 7), the average size was 8.5 modules. A list of the top 20 most common modules is presented, and explained, in Section 3 (Table 1). Additionally, programmers can create their own modules, referred to as subpipes that include existing pipes as subroutines. An example of this is shown later in this section.

To execute a pipe, the user loads the *information* view. When this page is loaded, the mashup is sent to Yahoo!'s servers, which interpret the mashup to fetch and manipulate the specified data sources and return the output back to the programmer. Fig. 1b shows the partial output of the mashup in Fig. 1a. The output of the pipe is a list of RSS items that reach the *pipe output* module; here, three items are shown. Shown is the title and description for each RSS item, but there are additional fields associated with the item, such as publication date, author, and URL. In this view, the user has the ability to embed the mashup feed in another website, *publish* it to the public repository, *clone* it to make a copy, or *edit source* and load the Pipes Editor.

2.2 Running Example

The mashup in Fig. 2a was retrieved from Yahoo's public repository. We added letter labels to serve as references to the modules to assist with the explanation. This mashup aggregates articles about online marketing from German websites and blogs, (e.g., http://internet-marketing-deutsch.blogspot.com/ in module F), and was selected for illustration due to the variety of smells it exhibits in a relatively small number of modules. We use the pipe in Fig. 2a as a running example to illustrate various concepts throughout this paper.

In Fig. 2a, five modules retrieve data from web sources: A, B, D, E, and F, each containing one field (the web data source). These data generating modules provide the inputs for the rest of the pipe modules to process. That is, these modules provide a list of RSS items, such as those shown in Fig. 1b, for the pipe to process. For each module in Fig. 2a, we show the output in Fig. 2c. For example, the output of module A, which gathers data from a web source, is represented as $out(A) : [a_1, a_2]$, where a_i is an RSS item. This means that the data source accessed by A provides a list of RSS items containing two items. For the sake of illustration, the lists from modules A and B are limited to two and four items, respectively. The lists for modules D, E, and F have arbitrary lengths of n_d, n_e, and n_f, respectively.

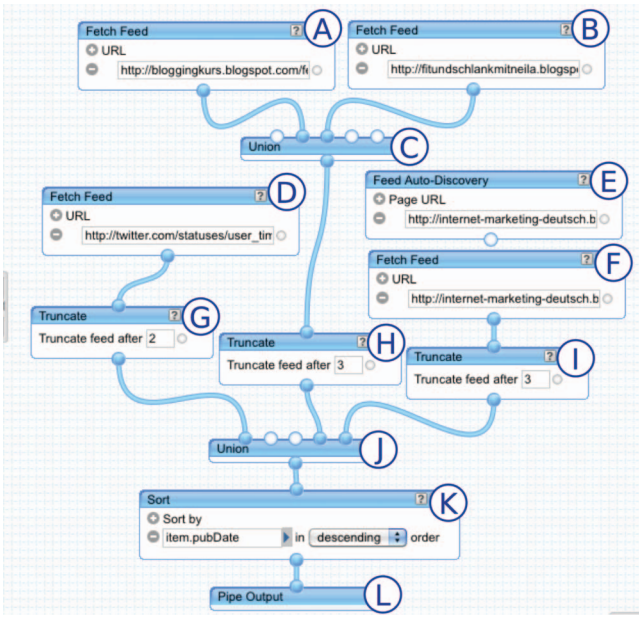
Modules D and F are wired directly into *truncate* modules G and I, respectively. *Truncate* modules retain the first n items to pass to the next module, where n is set by the field value. This behavior is illustrated in Fig. 2c. The output from D is a list $[d_1, d_2, \ldots, d_{n_d}]$, whereas the output from G is just the first two items, $[d_1, d_2]$. Similarly, the output from I has only three items from F.

The output from data generating modules A and B is aggregated through a path-altering *union* module, C, before feeding to *truncate* module H. That is, the output of C is the concatenation of the output from modules A and B, as shown in Fig. 2c. Then, module H limits the size of the list to just three items.

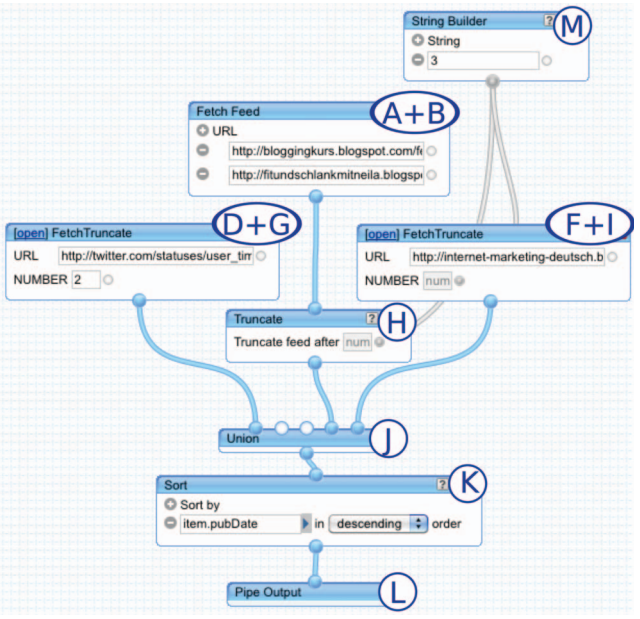
The outgoing data from modules G, H, and I are aggregated with a *union* module, J. Similarly to C, J concatenates the incoming lists from G, H, and I. The union module J feeds to a *sort* module, K, which reorders the list based on some property. In Fig. 2c, as we are using symbols to represent the RSS items, it was reordered arbitrarily. The sorted list from the output of K is wired to the pipe's *output* module, L. The output from L is what appears to the user when the pipe is executed, such as the executed pipe shown in Fig. 1b.

Fig. 2a indeed shows a functional pipe, yet it has several deficiencies that can be removed by transformations while preserving the underlying semantics. We identify some of those deficiencies here:

1. Module E is disconnected from the rest of the pipe (no connected wires) and can be removed without impacting the output of the pipe. This is evident when looking at the output from module L in Fig. 2c, as no RSS items from module E made it to the output.
2. The data produced by two generator modules, A and B, are immediately aggregated prior to any manipulation. Since generator modules can accommodate multiple fields, this redundancy can be removed by merging them into one module.
3. In two of the *truncate* modules, H and I, the string "3" specifies the number of data items to retain; if the user determines that this value represents the



(a) Original Pipe with Smells



(b) Completely Refactored Pipe

$$
\begin{aligned}
out(A) &: [a_1, a_2] \\
out(B) &: [b_1, b_2, b_3, b_4] \\
out(C) &: [a_1, a_2, b_1, b_2, b_3, b_4] \\
out(D) &: [d_1, d_2, \ldots d_{n_d}] \\
out(E) &: [e_1, e_2, \ldots e_{n_e}] \\
out(F) &: [f_1, f_2, \ldots f_{n_f}] \\
out(G) &: [d_1, d_2] \\
out(H) &: [a_1, a_2, b_1] \\
out(I) &: [f_1, f_2, f_3] \\
out(J) &: [d_1, d_2, a_1, a_2, b_1, f_1, f_2, f_3] \\
out(K) &: [a_2, b_1, f_2, d_1, d_2, a_1, f_1, f_3] \\
out(L) &: [a_2, b_1, f_2, d_1, d_2, a_1, f_1, f_3]
\end{aligned}
$$

(c) Original Pipe with Smells Example Execution

$$
\begin{aligned}
out(A+B) &: [a_1, a_2, b_1, b_2, b_3, b_4] \\
\\
out(D+G) &: [d_1, d_2] \\
out(H) &: [a_1, a_2, b_1] \\
out(F+I) &: [f_1, f_2, f_3] \\
out(J) &: [d_1, d_2, a_1, a_2, b_1, f_1, f_2, f_3] \\
out(K) &: [a_2, b_1, f_2, d_1, d_2, a_1, f_1, f_3] \\
out(L) &: [a_2, b_1, f_2, d_1, d_2, a_1, f_1, f_3] \\
out(M) &: \text{“3”}
\end{aligned}
$$

(d) Completely Refactored Pipe Example Execution

Fig. 2. Motivational example before and after refactoring.

same concept, then it can be abstracted into a separate module to facilitate and ensure consistency of future changes.

4. Two of the paths leading to module J, D to G and F to I (omitting the path from H to J), include a fetch module followed by a truncate module. These isomorphic paths can be abstracted into a separate pipe that can be included as a *subpipe* module, increasing the modularity of the pipe being analyzed.

These deficiencies are referred to as *code smells*. As we show later, code smells such as these can make pipes harder to understand and harder to maintain (Section 5). Given the popularity of the Yahoo! Pipes environment (over 90,000 users [4]), the fact that 46 percent of the pipes in our study were modified an average of 131 days after being created, and that 26 percent of the pipes in our study were modified after being made public, making future changes easier is likely a concern for mashup programmers. These smells can all be addressed by applying semantics-preserving transformations on the pipe, as we show next.

2.3 Addressing Deficiencies

The smells identified in Fig. 2a were addressed in the pipe in Fig. 2b. The impact on the output of the modules is reflected in Fig. 2d. Note that while individual modules may have different output, the output of the pipe, represented by $out(L)$, is the same in Figs. 2c and 2d. The changes made to the pipe in Fig. 2a are as follows:

1. Disconnected module E was removed. Depending on the server-side optimizations, this may increase performance as that RSS feed no longer needs to be fetched.
2. Generator modules A and B were merged to yield module $A+B$. This resulted in the removal of module C since it became ineffectual. The output of module $A+B$, shown in Fig. 2d, is the same as the output from module C in Fig. 2c.
3. A new module, M, was added to abstract the value “3” from modules H and I, which now receive their field values via wire from M. The added module M only provides values to *fields* in the pipe,

and not *modules*; this is reflected by $out(M) =$ "3" in Fig. 2d.

4. The isomorphic paths from D to G and from F to I were each replaced with a *subpipe* module that encapsulates that behavior, forming modules $D+G$ and $F+I$. (*Subpipes* are identified by the *[open]* link next to the modules' name.) The behavior of the pipe remains the same; the output of $D+G$ in Fig. 2d is the same as the output of G in Fig. 2c, and the same can be said for $F+I$ and I.

Through the transformation process, two of the original modules were removed, two modules were merged into one, two hard-coded fields are now abstracted in one place to ease future changes, and two new subpipes hide unnecessary details in the pipe making it smaller and less complex. These transformations are referred to as *refactorings*.

For most of the refactorings just described, our evaluation (described in Section 5) shows that users preferred the pipe without smells. The smells illustrated by this example were used in 10 tasks in which study participants were to select a smelly or a refactored pipe as being preferred (tasks 1, 2, 7, 8, 14, 15, 16, 17, 18, and 19). For nine of the 10 tasks, a majority preferred the refactored pipe, and this majority was significant at $\alpha = 0.1$ for six of the tasks. In one task (task 17) related to isomorphic paths, a majority preferred the smelly pipe, indicating that the transparency of behavior was preferred. In the example, this would mean the users preferred to see modules D and G separated, and F and I separated, as in Fig. 2a, rather than the combined modules $D+G$ and $F+I$ in Fig. 2b. According to the user comments, the behavior of the *subpipe* modules is not as clear as the behavior of a *fetch* module connected to a *truncate* module. In Section 5, we explain and explore the results of the study in greater detail, but first we introduce formal definitions for pipes and frequencies of occurrence of smells in the repository. Later, we describe our infrastructure for automatically detecting and removing the smells through refactoring (Section 7.2).

3 Definitions

In this section, we provide definitions that are used throughout the rest of the paper to define smells and refactoring transformations. Fig. 3 presents shorthand predicates used to simplify the presentation. Since a mashup represents a directional flow of data, we can represent a pipe-like mashup as a directed acyclic graph, where the modules are nodes and the wires are the edges that transmit data between the modules in a pipe.

Definition 1. *A* module *is a tuple* $(\mathcal{F}, \text{name}, \text{type})$*, containing a list of fields* $\mathcal{F}$ *indexed from 1 to* $|\mathcal{F}|$*, where* $\mathcal{F}[1]$ *is the first field in the list, a* name *assigned by the Pipes programming environment (e.g.,* fetch *or* truncate*), and a* type*, to be defined later in this section.*

Definition 2. *A* wire *is a tuple* $(src, dest, fld)$*, containing a module pointer to* src*, the source module of the wire, a module pointer to* dest *corresponding to the wire destination module, and a field pointer* fld *for the destination field, if one exists, and* $\emptyset$ *otherwise.*

$op(m)$	$m.type = op$
$op\_indep(m)$	$m.type = op.indep$
$op\_ro(m)$	$m.type = op.ro$
$setter\_str(m)$	$m.type = setter.string$
$path\_alt(m)$	$m.type = pathAlt$
$gen(m)$	$m.type = gen$
$output(m)$	$m.type = output$
$union(m)$	$m.name = union$
$split(m)$	$m.name = split$
$in\_wire(m,w)$	$w.dest = m \wedge w.fld = \varnothing$
$fld\_wire(m,w)$	$w.dest = m \wedge w.fld \neq \varnothing$
$out\_wire(m,w)$	$w.src = m$
$all\_fld\_wires(m)$	$\forall w \in \mathcal{W}\ (w.dest = m, fld\_wire(m,w))$
$joined\_by(m_i,m_j,w)$	$out\_wire(m_i,w) \wedge in\_wire(m_j,w)$
$subsequent\_mods(m_i,m_j)$	$\exists$ path $p(m_i,m_j \in p \wedge m_i \prec m_j)$
$btw\_mods(m_k,m_i,m_j)$	$\exists$ path $p(m_k,m_i,m_j \in p \wedge m_i \prec m_k \prec m_j)$
$conn\_to\_union(m_i,m_j)$	$\exists m_u \in \mathcal{M}\ (union(m_u) \wedge \exists w_i,w_j \in \mathcal{W}\ (joined\_by(m_i,m_u,w_i) \wedge joined\_by(m_j,m_u,w_j)))$
$same\_n\_flds(m_i,m_j)$	$\lvert m_i.\mathcal{F}\rvert = \lvert m_j.\mathcal{F}\rvert$
$same\_fld\_vals(m_i,m_j)$	$same\_n\_flds(m_i,m_j) \wedge$ for $k = 1 \ldots \lvert m_i.\mathcal{F}\rvert, m_i.\mathcal{F}[k] = m_j.\mathcal{F}[k]$
$same\_val(f_i,f_j,bool)$	$f_i.wireable = bool \wedge f_j.wireable = bool \wedge f_i.value = f_j.value \wedge f_i.value \neq$ ""

Fig. 3. Shorthand for common predicates used in the definitions, smells, and refactorings.

Definition 3. *A* field *is a tuple* $(\text{wireable}, \text{value})$ *containing a function* $\text{wireable}(\mathcal{F}) \rightarrow \{\text{true} \mid \text{false}\}$ *indicating whether or not that field can be set by an incoming wire, and a* value *that contains a string-representation of the field's content.*

Definition 4. *A* pipe *is a graph,* $PG = (\mathcal{M}, \mathcal{W}, \text{owner})$,[1] *containing a set of modules* $\mathcal{M}$*, a set of wires* $\mathcal{W}$*, and a function* $\text{owner}(f) \rightarrow \mathcal{M}$ *assigning every field* f *to exactly one module. The wires are constrained such that* $\forall w \in \mathcal{W}((w.src \in \mathcal{M}) \wedge (w.dest \in \mathcal{M}) \wedge (w.src \neq w.dest) \wedge (w.fld \neq \emptyset \vee w.fld \in (\text{w.dest}).\mathcal{F}))$ *(no cyclic wires, source and destination are within* PG*, and if the wire goes to a field that field is within* w.dest*). Every pipe must also contain exactly one module named output.*

For example, Fig. 2a shows a pipe with $|\mathcal{M}| = 12$, modules $A, C \in \mathcal{M}$ connected by wire $w \in \mathcal{W}$, where $w = (A, C, \emptyset)$. Module K has two nonwireable fields, (e.g., $K.\mathcal{F}[1] =$ (false,"item.pubDate")), whereas module G has one wireable field, $G.\mathcal{F}[1] =$ (true, "2").

Definition 5. *A pipe path is a sequence of* n *connected modules* m_i *such that* $\forall m_i((0 \leq i < n-1) \rightarrow \exists w \in \mathcal{W}(out\_wire(m_i, w) \wedge in\_wire(m_{i+1}, w) \wedge (m_i \prec m_{i+1})))$*. For notational convenience, the path length is defined by* $p.length$*, the first module in the path can be accessed by* $p(first)$*, and is closer to the data source. The last module in the path by* $p(last)$*, and is*

1. Our previous definition [1] for a pipe was $PG = (\mathcal{M}, \mathcal{W}, \mathcal{F}, owner)$; since fields are always contained within modules, we removed the redundancy of including $\mathcal{F}$ in the pipe definition.

closer to the output. Using the example in Fig. 2a, the modules and wires from G to K form a path p, where $p.length = 3$, $p(first) = G$, and $p(last) = K$.

As part of the module definition, we classify modules by their type, where the type is defined based on the module's impact on the data that flows through the mashup. Types are meant to classify modules at a higher level of abstraction than the names, similar to a type hierarchy. These types are intended to be general across mashup applications, as we show implicitly by later mapping some of our definitions onto the DERI pipes environment (Section 10.1).

The module types are nonoverlapping in that a module is assigned exactly one type. However, there are some orthogonal subtypes used with the operator and setter modules, as we describe. The *type* for a module $m \in \mathcal{M}$ is defined as follows:

1. *Generator module.* $m.type = gen$ if $\exists f \in m.\mathcal{F}$ such that f refers to an external data source (e.g., an RSS feed, another pipe). This is the only module type that provides a list of items for other modules in the pipe to process. From Fig. 2a, modules A, B, D, E, and F are *generator* modules, and from Fig. 2b, modules $A + B$, $D + G$, and $F + I$ are generator modules. The $subpipe$ modules are classified as generator modules because each $subpipe$ is itself a complete pipe, where the output of a $subpipe$ modules is the output from the pipe it abstracts.
2. *Setter module.* $m.type = setter$ if m sets a value that will be used to parameterize the pipe; all the wires outgoing from a setter module have a field as the destination. That is, $\forall w \in \mathcal{W}((w.src = m) \rightarrow (w.fld \neq \emptyset))$. A setter module is said to be a *string-setter* module, if m sets a string ($m.type = setter.string$). In Fig. 2b, M is a *string-setter* module. A setter module is said to be a *user-setter* module if the user may be queried to set a parameter when the pipe is executed ($m.type = setter.user$). A *user-setter* module is connected to the pipe in the same way as M in Fig. 2b.
3. *Path-altering module.* $m.type = pathAlt$ if m either joins multiple paths, as in a union, or diverts one path into multiple paths, as in a split. That is, $\exists w_i, w_j \in \mathcal{W}(w_i.src = m \wedge w_j.src = m \wedge w_i.fld = \emptyset \wedge w_j.fld = \emptyset)$, as in a split (semantically works like a copy), or $\exists w_i, w_j \in \mathcal{W}(w_i.dest = m \wedge w_j.dest = m \wedge w_i.fld = \emptyset \wedge w_j.fld = \emptyset)$, as in a union.

   Modules C and J in Fig. 2a are *union* modules; the input lists are concatenated together to create the output list. A *split* is like an upside-down *union*, in which the outgoing wires contain copies of the incoming wire. In the Yahoo! Pipes language, these two modules are the only path-altering modules.
4. *Operator module.* $m.type = op$ if m performs a manipulation operation (e.g., sorting, removing, renaming) on the list of items passed in via wire.

   An operator module o can be subtyped across two orthogonal dimensions: o is said to be *read-only* ($op.ro$) if it does not modify the content of items in the input list (e.g., filtering a list based on the items' titles), and *read-write* ($op.rw$) if it can modify the content of list items (e.g., appending a string to each item's title). Modules G, I, and K in Fig. 2a are $op.ro$ modules. A *rename* module that augments title values of the items would be subtyped as $op.rw$. Second, o is said to be *order-independent* ($m.type = op.indep$) if the set of items in the output from the module is the same regardless of the order of the items passed into it (e.g., reordering, renaming, or removing list items), or o is *order-dependent* ($m.type = op.dep$) if the outcome depends on the order of the items passed into it (e.g., the *truncate* module that only outputs the first n items in a list). In Fig. 2a, module K, a $sort$ module, is $op.indep$ and modules G and I, $truncate$ modules, are $op.dep$.
5. *Output module.* $m.type = output$ if m has the following property: $m.\mathcal{F} = \emptyset \wedge \exists w \in \mathcal{W}(w.dest = m \wedge w.fld = \emptyset) \wedge (\nexists w \in \mathcal{W}(w.src = m))$. Every pipe contains exactly one output module. In Figs. 2a and 2b, module L is an *output* module.

TABLE 1
Most Common Module Frequencies and Types in Yahoo! Pipes

Frequency	$m.name$	$m.type$	Figure 2
96,026+	$output$	$output$	L
96,026	$fetch$	gen	A,B,D,F,$A + B$
45,251	$filter$	$op.indep \wedge op.ro$	
36,480	$sort$	$op.indep \wedge op.ro$	K
31,145	$union$	$pathAlt$	C, J
22,775	$regex$	$op.indep \wedge op.rw$	
21,573	$loop$	$op.indep \wedge op.rw$	
16,812	$unique$	$op.indep \wedge op.ro$	
15,924	$rename$	$op.indep \wedge op.rw$	
15,418	$textinput$	$setter.user$	
14,453	$truncate$	$op.dep \wedge op.ro$	G, H, I
13,055	$urlbuilder$	$setter.string$	
12,252	$fetchsitefeed$	gen	
11,746	$fetchpage$	gen	
11,374	$strconcat$	$setter.string$	M
7,935	$subpipe$	gen	$D + G$, $F + I$
7,138	$rssitembuilder$	gen	
5,651	$createrss$	$op.indep \wedge op.rw$	
5,409	$flickr$	gen	
5,399	$ysearch$	gen	

To provide some more background on the Yahoo! Pipes language, the top 20 most frequently used modules are shown in Table 1. The *Frequency* column lists the number of public pipes containing that module as of June 2013, the $m.name$ column identifies the module name given by the Yahoo! Pipes environment, and $m.type$ classifies each according to the definition of types just presented. The last column, Fig. 2, identifies the modules in Fig. 2 that map to the modules listed. Other than the $output$ module that appears in every pipe, the $fetch$ module is the most common, appearing in 96,026 publicly available pipes; it is of type gen. The $filter$ appears in nearly half as many, 45,251 pipes, and is of type $op.indep \wedge op.ro$.

4 Code Smells

To ascertain smells relevant to pipe-like mashups, we collected candidate smells similar to those defined for professional programmers (as per the references in Section 9), defined smells based on errors and unnecessary complexity reported in the users' newsgroups, and identified others by

observing the rich sample of over 8,000 pipes we gathered for analysis (Section 7 describes the selection criteria). Utilizing this sample, the list of candidate smells was iteratively refined as we found smells that were either not directly applicable to the domain (e.g., refactoring for concurrency, cleaning up class hierarchies, modifying conditionals for simplicity, introducing design patterns, pulling up constructor bodies) or not common enough (< 5 percent) to warrant their consideration (e.g., *authentication needed*, which occurs when a URL returns a 401 error).

With the study infrastructure described in Section 7.2, we analyzed a sample of pipes to determine the prevalence of smells among pipes in the community. For each smell defined in this section, we report the percentage of pipes that contain the smell. For example, Smell 1: *Noisy Module* appears in 28 percent of the pipes we studied. Overall, 81 percent of the pipes were afflicted with one or more smell.

Each smell is defined as a predicate in the context of a pipe represented as a graph $PG = (\mathcal{M}, \mathcal{W}, owner)$. Using Smell 2: *Unnecessary Module* as an example, we identify an instance of this smell for module m if the predicate $\exists n \in \mathcal{M}(m \neq n \wedge output(n) \wedge !subsequent\_mods(m, n))$ evaluates to true. This could occur if m is disconnected from all other modules (i.e., it has no incoming of outgoing wires, like Module E in Fig. 2). In that case, the output module, n, will not be a subsequent module from m, and hence the predicate will evaluate to true, identifying an instance of the smell.

Next, we look at the smell definitions and then Section 5 presents a detailed account of user preferences regarding the defined smells.

4.1 Laziness Smells

This category of smells was inspired in part by the "Lazy Class" smell, which identifies classes, components, or methods that "do not do enough" [9]. These smells identify pipes that contain modules or fields that do not contribute to the output of the pipe, making it unnecessarily complex or potentially faulty.

Smell 1. Noisy module (28 percent)—A module that has unnecessary fields, making the pipe harder to read and less efficient to execute. Module $m \in \mathcal{M}$ is considered noisy if:

Case 1.1. Empty field:

$$(gen(m) \vee setter\_str(m)) \wedge \exists f \in m.\mathcal{F}(f.value = \text{``''}).$$

This describes a blank field in a generator or string-setter module. If a generator has no specified URL, then it generates no output and does not contribute and content to the pipe.

Case 1.2. Duplicated field:

$$\exists f_i, f_j \in m.\mathcal{F} \mid f_i \neq f_j \wedge same\_val(f_i, f_j, true).$$

This describes the case when two fields in a single module have the same value. For example, if a $fetch$ module contained the same URL in two different fields, then this smell would be present.

Smell 2. Unnecessary module (13 percent)—A module whose execution does not affect the pipe's output, adding unnecessary complexity. Module $m \in \mathcal{M}$ is considered unnecessary if:

Case 2.1. Cannot reach output:

$$\exists n \in \mathcal{M}(m \neq n \wedge output(n) \wedge !subsequent\_mods(m, n)).$$

This describes a module that is disconnected from the pipe or that does not contain an outgoing wire such that there exists a path from the module to the output (i.e., a dangling module). To illustrate, consider the smell definition and Fig. 2a; m from the definition maps to E in Fig. 2a and n maps to L. Since n is the output, $m \neq n$, and there does not exist a path between m and n (per the definition of $subsequent\_mods$ in Table 3), this smell is present in the pipe shown in Fig. 2a.

Case 2.2. Ineffectual path altering:

$$\begin{aligned} &path\_alt(m) \\ &\wedge\ \exists_1 w_i \in \mathcal{W}(in\_wire\ (m, w_i)) \\ &\wedge\ \exists_1 w_j \in \mathcal{W}(out\_wire\ (m, w_j)). \end{aligned}$$

This describes a path altering module with exactly one input wire and exactly one output wire. In the transformation from Fig. 2a to Fig. 2b, modules A and B consolidate to $A + B$ and module C disappears. After $A + B$ has replaced A and B, then this smell is present where m maps to C. Since C is a path-altering module that has exactly one wire in, and one wire out, this smell was present. Eliminating this smell was an intermediate step in the transformation from Fig. 2a to Fig. 2b and is therefore not explicitly shown.

Case 2.3. Inoperative module:

$$(setter\_str(m) \vee op(m) \vee gen(m)) \wedge (m.\mathcal{F} = \emptyset).$$

This describes a generator, operator, or setter module that contains all blank fields, and hence has no behavior. Where Smell 1.1: *Empty Field* describes a single instance of an empty field, this smell describes a module that has all empty fields.

Case 2.4. Unnecessary redirection:

$$setter\_str(m) \wedge (|m.\mathcal{F}| = 1) \wedge all\_fld\_wires(m).$$

A module that is a setter module with exactly one field that receives its value via wire provides unnecessary redirection. To illustrate, consider module M in Fig. 2b. Hypothetically, if the field in M received its value via wire from another module, then M would meet the criteria for this smell since it is a setter, it has one field, and its one field would receive its value via wire.

Case 2.5. Swaying module:

$$(path\_alt(m) \vee op(m)) \wedge\ \nexists w \in \mathcal{W}(in\_wire(m, w)).$$

These are modules that do not provide any data to the pipe and do not receive any incoming data. For example, an operator module that does not have a generator feeding it items would be swaying.

Smell 3. Unnecessary abstraction (12 percent)—A setter module that always performs the same operation on constant field values (fields that are not wired), and that only feeds a value to one destination, may be unnecessarily abstract. Unlike Smell 2.4: *Unnecessary redirection*, which describes a module that has exactly one wired field and any number of outgoing fields, this string setter module can only have one destination and none of its fields are wired. Module $m \in \mathcal{M}$ is unnecessary if:

$$setter\_str(m) \wedge \exists_1 w_i \in \mathcal{W}(out\_wire(m, w_i)) \\ \wedge \nexists w_j \in W(fld\_wire(m, w_j)).$$

To illustrate, consider module M in Fig. 2b. Hypothetically, if there was only one outgoing wire from M, then it would match this smell because it would have one outgoing wire and none of the fields in M are wired.

4.2 Redundancy Smells

Duplicated code has been referred to as the worst smell in programs written by professionals [9]. These redundancy smells identify pipes that have duplicated fields, modules, or paths. Redundancies in pipes bloat the modules and the pipe structure, add unnecessary complexity, and make pipe understanding and maintenance more difficult.

Smell 4. Duplicate strings (32 percent)—A constant string that is used in at least n wireable fields in at least two modules. Given $n = 2$, fields are marked as duplicates if:

$$\exists f_i, f_j \in \bigcup_{m \in \mathcal{M}} m.\mathcal{F}((owner(f_i) \neq owner(f_j)) \\ \wedge\ same\_val(f_i, f_j, true)).$$

This is similar to Smell 1.2: *Duplicate field* except the fields must be in different modules and have the opportunity to receive values via wire. For example, in Fig. 2a the truncate modules H and I have a duplicate string "3." This means that f_i maps to the field in H, and f_j maps to the field in I. Since $((owner(f_i) \neq owner(f_j))$, which is equivalent to saying $H \neq I$, and f_i and f_j hold the same value (i.e., "3"), then the pipe shown in Fig. 2a has this smell.

Smell 5. Duplicate modules (23 percent)—Operator modules appearing in certain patterns may be redundant and candidates for consolidation. Modules $m_i, m_j \in \mathcal{M}$ are considered duplicates if $m_i.name = m_j.name$ and

Case 5.1. Consecutive redundant modules:

$$(op\_indep(m_i) \vee path\_alt(m_i) \\ \wedge\ \exists w_j \in \mathcal{W}\ (joined\_by(m_i, m_j, w_j)).$$

This describes two order-independent or path-altering modules, such as filters or unions, that are incident to one another in the pipe. As an example, consider the pipe shown in Fig. 2a. Hypothetically, if H was removed and C connected directly to J, this smell would be present. In that situation, m_i would map to C and m_j would map to J. In the absence of H, there would be a wire, w_j joining C to J. Fig. 5, described in Section 6.2, shows three different instances of this smell.

Case 5.2. Identical subsequent operators:

$$op\_indep(m_i) \wedge op\_ro(m_i) \wedge same\_fld\_vals(m_i, m_j) \\ \wedge\ subsequent\_mods(m_i, m_j) \\ \wedge\ \forall m_k \in \mathcal{M}(btw\_mods(m_k, m_i, m_j) \\ \wedge\ (union(m_k) \vee (op\_indep(m_k) \wedge op\_ro(m_k))).$$

This describes two identical order-independent, read-only operator modules that exist along the same path in a pipe and all the modules along that path are order-independent or union modules. In this way, the module farther from the output is unnecessary since it performs the same operations as the one closer. As an example, consider two sort modules that both order a list based on publication date. This operation only needs to happen once because a sorted list does not need to be resorted. Fig. 6 shows an example of this smell.

Case 5.3. Joined generators:

$$gen(m_i) \wedge gen(m_j) \wedge conn\_to\_union(m_i, m_j) \\ \wedge\ out\_wire(m_i, w_i) \wedge out\_wire(m_j, w_j).$$

This describes two generators that are directly connected to a union, and names the outgoing wires from each for use in the associated refactoring transformation (Section 6). This smell is present in the pipe in Fig. 2a, where m_i maps to A, m_j maps to B, and both are connected to a union, C. Another example of this smell is shown in Fig. 7.

Case 5.4. Identical parallel operators:

$$op\_indep(m_i) \wedge op\_indep(m_j) \\ \wedge\ same\_fld\_vals(m_i, m_j) \\ \wedge\ conn\_to\_union(m_i, m_j) \\ \wedge\ \exists m_k, m_l \in \mathcal{M}\ \exists w_i, w_j, w_k, w_l \in \mathcal{W} \\ (out\_wire(m_i, w_i) \wedge out\_wire(m_j, w_j) \\ \wedge\ joined\_by(m_k, m_i, w_k) \wedge joined\_by(m_l, m_j, w_l) \\ \wedge\ (gen(m_k) \vee union(m_k)) \wedge (gen(m_l) \vee union(m_l))).$$

This describes two order-independent operators that are parallel in the pipe with structures that could be represented with a single pipe path by consolidating modules like the generators. Three instances of this smell are shown in Fig. 8.

Smell 6. Isomorphic paths (7 percent)—Nonoverlapping paths with the same modules performing the same operations may be missing a chance for abstraction. Two paths p and p' are isomorphic if:

$$(p.length = p'.length) \wedge (p \cap p' = \emptyset) \\ \wedge\ gen(p(first)) \wedge gen(p'(first)) \\ \wedge\ \forall m_n \in p, \forall m'_n \in p'((0 \leq n < p.length) \\ \rightarrow ((m_n.name = m'_n.name) \\ \wedge\ same\_n\_flds(m_n, m'_n) \\ \wedge\ (\forall f \in m_n.\mathcal{F}, \forall f' \in m'_n.\mathcal{F}((f.wireable = false) \\ \rightarrow same\_val(f, f', false))))).$$

An example is shown in Fig. 2a, where p consists of the path from D to G (i.e., $p(first) = D$ and $p(last) = G$) and p' consists of the path from F to I. This definition can be generalized to n isomorphic paths by defining a new p' for each subsequent isomorphic path.

4.3 Environmental Smells

Inspired by the pervasive use of invalid and unsupported data sources and modules by pipes in the Yahoo! Pipes repository, these smells identify pipes that have not been updated in response to changes to the external environment. A pipe containing a module no longer maintained by the Pipes language or a field that references an invalid external source exhibits an environmental smell that may cause a failure.

Smell 7. Deprecated module (18 percent)—A module that is no longer supported by the pipe environment. Given $Supported_{\mathcal{M}}$, a pipe contains this smell if: $\exists m \in \mathcal{M}$ such that $m.name \notin Supported_{\mathcal{M}}$.

At least four modules have been deprecated in the Yahoo! Pipes environment from 2007 to 2012.

Smell 8. Invalid Sources (14 percent)—An external data source $es \in ExternalSources$ is invalid if n consecutive attempts to retrieve data from it report errors, such as `404 Not Found`. Given $n = 1$, a pipe presents this smell when $\exists f \in \mathcal{F}$ that refers to an invalid es. Typically, this smell shows up in the generator modules where $owner(f).type = gen$. Such fields add unnecessary size to the pipe.

4.4 Population-Based Smells

The previous smells focused on individual pipes. Population-based smells, on the other hand, rely on the community knowledge captured in the public repository to discover patterns that have been commonly employed in highly reused pipes or that are common among all pipes. Pipes using alternative module structures to implement such patterns are considered smelly because they may take more time to understand and potentially discourage reuse of those pipes across the community.

Smell 9. Nonconforming module orderings (19 percent)—Given a community prescribed order for read-only, order-independent operator modules appearing in a path of length n, a pipe with a path including such modules but in a different order may unnecessarily increase the difficulty for other users to understand and adopt the pipe. By performing a frequency analysis of the pipes in a repository, we obtain a pool of commonly observed paths that we call prescribed paths, $PPres$, and consider path p to be nonconforming if:[2]

$$\begin{aligned}\forall\, m \in p\ ((op\_indep(m) \wedge m.type = op.ro)\\ \wedge\ \exists p' \in PPres\ ((order(p) \neq order(p'))\\ \wedge\ (bag(p) = bag(p')))).\end{aligned}$$

For example, an instance of this smell would be identified if p and p' have the same bag of modules, yet their ordering is different (e.g., $p = [filter, sort, filter]$, $p' = [sort, filter, filter]$, and p' is a prescribed path from $PPres$). Defining $PPres$ requires the identification of the sample of the population from which the prescribed paths are to be derived and the bounding of the path length to be considered. Section 7.1 describes how we detected instances of this smell in the population.

Smell 10. Global isomorphic paths (6 percent)—Building on the isomorphic path smell (Smell 6: *Isomorphic Paths*), we extend the scope of the smell to paths appearing in multiple pipes. Global isomorphic paths represent missed opportunities for a community to reuse the work of its contributors, and make it harder to understand pipes due to the lack of abstraction of commonly occurring paths. Given a pool of prescribed global paths $PGPaths$, a pipe PG has this smell if:

$$\exists p \in PG, \exists p' \in PGPaths(p' \text{ is isomorphic to p}).$$

As with the previous smell, generating $PGPaths$ requires identification of the population sample from which the paths are derived and a threshold of path frequency for it to be considered global. Using Fig. 2a as an example, if the paths from D to G and from F to I are part of the $PGPaths$, then this smell would be present in that pipe. Section 7.1 provides the implementation details for detecting this smell.

2. We use $bag(p)$ as a data structure representing the names and counts of modules in path p, and $order(p)$ to represent the ordering of names in path p from first to last.

5 User Preferences and Smelly Pipes

From a researcher's perspective, code smells are indicative of deficiencies in source code, and refactorings are intended to make the code better with respect to some property, such as understandability or maintainability. Yet, for web mashups, it is not clear if code smells in programs matter to programmers. Toward this end, we designed two experiments. The first experiment aims to determine if the programmers prefer pipes with or without smells, and the second aims to determine if smelly pipes are harder to understand than clean pipes. We address the following research questions:

- *RQ1:* Are pipes with smells more or less desirable than pipes without such characteristics? (study 1)
- *RQ2:* Are pipes with smells more or less understandable than pipes without such characteristics? (study 2)

5.1 Experimental Design

The experiments that explore these research questions were split into tasks with a random assignment of subjects to tasks. Each task has two semantically, but not syntactically, equivalent pipes. As we are interested in the impact of smells, the treatment is a smell applied to a pipe (object). For each task, one pipe is treated with a smell and the other is not, providing coverage for all the smells defined in Section 4 and a variety of pipe structures. The dirty pipes were gathered from the community artifacts, and the clean pipes were generated by applying a refactoring transformation (defined in Section 6). The following sections describe the artifacts used in the study, implementation, and participants.

5.1.1 Experimental Artifacts

Sixteen pairs of pipes (one clean/refactored and one dirty/smelly) were used in the study, and each pair is described in Table 2. Some of the pipe pairs were used by multiple experimental tasks. The *Artifacts* column enumerates the pipe pairs for reference. The *Smell* column indicates the smell that is present in the dirty pipe and absent in the clean pipe; we include the smell number from Section 4 for easy reference. The *Purpose* of each pipe is described to show the diversity in the tasks programmers can accomplish within this domain, from getting job ads to finding pictures uploaded by friends. These pipes were selected based on their smells to provide coverage of all smells defined in Section 4.

To illustrate the size and complexity of each pipe, we flatten the pipe to a string representation. The *Dirty Pipe* and *Clean Pipe* columns describe the structures of the pipes using an extended parallel-serial graph representation [35]. Parentheses and commas denote parallel paths (e.g., `(x, y)` represents two parallel paths, `x` and `y`, separated by a comma) and spaces denote serial paths (e.g., `x y z` is a serial sequence of `x-y-z`). The *setter* modules, which can have multiple output wires and be connected to nearly any module in the pipe, break the parallel-serial structure, so we modify the representation by treating these modules as

TABLE 2
Study Artifact Descriptions

Artifacts	Smell	Purpose	Dirty Pipe	Clean Pipe
1	5: Duplicate Modules (joined generator)	Aggregate news, limit 15 items total	(fetch, fetch, fetch, fetch, fetch) union truncate output	fetch truncate output
2	4: Duplicate Strings	Aggregate news, limit three items each feed	((fetch truncate),(fetch truncate), (fetch truncate), (fetch truncate), (fetch truncate)) union output	((fetch, textinput0) truncate,(fetch, textinput0) truncate, (fetch, textinput0) truncate, (fetch, textinput0) truncate, (fetch, textinput0) truncate) union output
3	5: Duplicate Modules (consecutive redundant operator)	Modify titles of items and return new results	(fetch regex regex sort, dateBuilder) filter output	(fetch regex sort, dateBuilder) filter output
4	8: Invalid Source	Aggregate and sort community news	(fetch, fetch, fetch, fetch, fetch) union sort output	(fetch, fetch, fetch) union sort output
5	7: Deprecated Modules	Obtain Flickr images for each item	fetch contentAnalysis forEachReplace(Flickr) output	fetch loop(termextractor) loop(flickr) output
6	1: Noisy Module (duplicated field)	Get job ads for specific job titles	fetch - filter - output	fetch - filter - output
7	10: Global Isomorphic Paths	Get unique items from several blogs	fetch unique unique sort output	subpipe sort output
8	9: Module Ordering, 2: Unnecessary Module	Get unique news items about "android"	fetch filter union unique output	fetch unique filter output
9	8: Invalid Source	Retrieve pictures uploaded by certain authors	fetch sort filter truncate output	fetch sort filter truncate output
10	7: Deprecated Modules	Search for wineries near Yountville, CA	yahooLocal foreachAnnotate(flickr) sort output	yahooLocal loop(flickr) sort output
11	6: Isomorphic Paths	Perform a job search within a zip code	((textinput0, textinput1) stringbuilder urlbuilder fetch rename, (textinput0, textinput1) stringbuilder urlbuilder fetch rename, (textinput0, textinput1) stringbuilder urlbuilder fetch rename, (textinput0, textinput1) googleBase rename) union sort output	((textinput0, textinput1) subpipe, (textinput0, textinput1) subpipe, (textinput0, textinput1) subpipe, (textinput0, textinput1) googleBase rename) union sort output
12	4: Duplicate Strings	Get news about Cerner	(yahooSearch, googleBase, fetch) union filter sort unique output	((stringBuilder0 yahooSearch, stringBuilder1 googleBase, fetch) union, stringBuilder0, stringBuilder0, stringBuilder1, stringBuilder1, stringBuilder2, stringBuilder2) filter sort unique output
13	6: Isomorphic Paths	Get twitter feeds from specific users	((fetch truncate),(fetch truncate), (fetch truncate), (fetch truncate), (fetch truncate)) union unique filter sort output	(subpipe, subpipe, subpipe, subpipe, subpipe) union unique filter sort output
14	4: Duplicate Strings	Get news about e-learning	((stringBuilder1 urlBuilder fetch, stringBuilder2 urlBuilder fetch, stringBuilder3 urlBuilder fetch) union, textinput0) filter sort output	((stringBuilder0 stringBuilder1 urlBuilder fetch, stringBuilder0 stringBuilder2 urlBuilder fetch, stringBuilder0 stringBuilder3 urlBuilder fetch) union, textinput0) filter sort output
15	6: Isomorphic Paths	Get NY Jets news	(fetchSiteFeed, (fetch filter), (fetch filter), (fetch filter)) union sort unique output	(fetchSiteFeed, subpipe, subpipe, subpipe) union sort unique output
16	5: Duplicate Modules (joined generator), 3: Unnecessary Abstraction	Search for deals from shopping websites	(((urlBuilder, urlBuilder) fetch, urlBuilder fetch, urlBuilder fetch) union, textinput) filter output	(urlBuilder fetch, textinput)filter output

symbolic. That is, to capture all the outgoing connections for each *setter* module, we assign each a symbolic name (e.g., *textinput0* and *textinput1* would be symbolic names assigned to two *textinput* modules), and serially attach the symbolic names to each module to which it is connected. To illustrate, consider the pipe in Fig. 2a. Its parallel-serial graph is represented as `(fetch truncate, (fetch, fetch) union truncate, fetch truncate) union sort output`, and the pipe in Fig. 2b is represented as `(subpipe, (fetch, stringBuilder0) truncate, stringBuilder0 subpipe) union sort output`. In Fig. 2b, the *setter* module, `stringBuilder`, has two destinations, a `truncate` module and a `subpipe` module. Thus, it is assigned a symbolic name, `stringBuilder0` and serially attached to each.

5.1.2 Study Implementation

This study was implemented using Amazon's Mechanical Turk [25], a service advertised as a "marketplace for work that requires human intelligence." There are two roles in Mechanical Turk, a *requester* and a *worker*. The requester is the creator of a *human intelligence task*, or *HIT*, which is intended to be a small, goal-oriented task that can be accomplished in less than 60 seconds. This fits our study

TABLE 3
RQ1 Task Summaries

Task	Smell	Artifacts	Refactoring	User Goal
1	Redund.: 5	1	Consolidation	To Understand
19	Redund.: 5	1	Consolidation	To Maintain
3	Redund.: 5	3	Consolidation	To Understand
2	Redund.: 4	2	Abstraction	To Maintain
17	Redund.: 4	2	Abstraction	To Understand
15	Redund.: 4	12	Abstraction	To Maintain
18	Redund.: 4	14	Abstraction	To Maintain
14	Redund.: 6	11	Abstraction	To Maintain
16	Redund.: 6	13	Abstraction	To Maintain
7	Redund.: 6 Pop.: 10	7	Abstraction, Standardize	To Understand
8	Laziness: 2 Pop.: 9	8	Reduction Standardize	For Others to Understand
6	Laziness: 1	6	Reduction	To Maintain
4	Environ.: 8	4	Deprecation	To Maintain
10	Environ.: 8	9	Deprecation	To Understand
12	Environ.: 8	9	Deprecation	To Maintain
5	Environ.: 7	5	Deprecation	For Others to Understand
11	Environ.: 7	10	Deprecation	To Understand
13	Environ.: 7	10	Deprecation	To Maintain

structure described previously. The worker completes the HIT and gets paid for their work, if satisfactory. Each task we defined for this study was implemented as a HIT, and users were paid $0.25 per HIT completed.

Participants were given a maximum of 60 minutes to complete each HIT and the study was launched in two phases. The first and initial phase ran from April 28 to May 13, 2010, and the second phase ran from July 5 to September 10, 2011. This second phase was launched to explore further some questions that remained after the initial phase, specifically related to abstraction smells. The workflow for a user participating in our study is as follows: A user must first create an account in Mechanical Turk and then locate our HITs by searching. Next, they may read some tutorial information and must take a qualification test. This test is used to control for participant quality, collect demographic information about the participants (e.g., education level), and obtain Institutional Review Board[3] consent, per our institution's policy. Once a user submitted a qualification test, it was graded as per our specification. A passing score allowed the user to complete any of the HITs in this study. Retakes on the qualification exam were not permitted in the event of a failing score.

5.1.3 Study Participants

Refactoring code is a task that was originally proposed and studied in the context of professional programmers. In this work, we adapt and extend refactoring to an end-user programming language in the web mashup domain. We solicited participation in the study from both end-user programmers and degreed programmers.

We classified participants using their responses to a survey question about their education level relating to computer science and related fields, which was a part of the qualification test. Users who reported to have a *degree in computer science or related field* were classified as *degreed users*, while the rest were classified as *end users*.

3. The Institutional Review Board, or IRB, is an organization that reviews and authorizes study protocol for experiments involving human subjects to protect the volunteer subjects. The approval number for this project is: UNL IRB# 20110410792 EX.

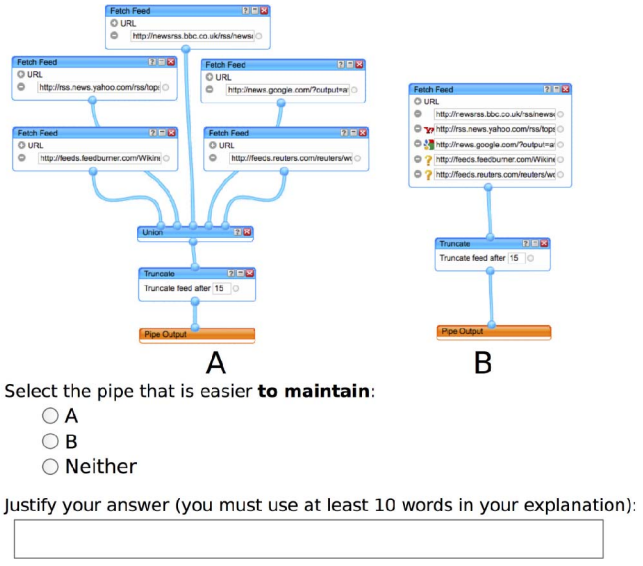


Fig. 4. Illustration of Task 19.

The survey and qualification test were completed by 258 subjects and 135 (52 percent) received a passing score of 50 percent or more. A total of 61 subjects participated in the study, and the average qualification test score for those participants was 74 percent. Of those, 29 (48 percent) were classified as *degreed users*, with an average score of 72 percent on the qualification test. The remaining 32 (52 percent) participants were classified as *end users* and scored an average of 75 percent on the qualification test. The participants provided a total of 366 data points across all tasks, and the average participant completed approximately six tasks.[4]

5.2 Study Results

We now describe the approach for addressing each research question, and the results.

5.2.1 RQ1: Are Pipes with Smells Less Preferable?

For this experiment, we defined tasks using the pipe pairs from Table 2. In each task, the user was shown the clean pipe and the dirty pipe side-by-side and asked to select the pipe that is preferable, given some goal (i.e., *to understand*, *to maintain*, or *for others to understand*). The first goal, understandability, explores the clarity of the clean pipe with respect to the dirty version. The second goal, maintainability, explores the effort of the user to update the pipe in the future; the third goal has an understandability and community focus. Using radio buttons, participants selected the first, second, or neither pipe as being preferable and justified their answer with a free-form response.

The experiment to explore this research question was split into 18 tasks. Artifact sets 1-14 from Table 2 were used in these tasks, which are described in further detail in Table 3 (eight tasks (1-8) were evaluated in the first phase, and 10 tasks (10-19) were evaluated in the second phase). The *Task* column indicates the task number (for reference), and the *Smell* describes the type of smell involved with the

4. At $\alpha = 0.1$ or lower, there were no significant differences between the results end users and degreed users, so we aggregate the responses.

TABLE 4
RQ1 Tasks and Results Breakdown

Task	All Results			$H_0 : \pi_c \leq 0.5$	$H_0 : \pi_d \leq 0.5$	$H_0 : \pi_n \leq 0.5$
	Clean	Dirty	Neither	$H_a : \pi_c > 0.5$	$H_a : \pi_d > 0.5$	$H_a : \pi_n > 0.5$
1	**15**	2	0	$p = 0.0018$**	$p = 0.9982$	$p = 0.9999$
2	**15**	2	0	$p = 0.0018$**	$p = 0.9982$	$p = 0.9999$
3	**9**	3	5	$p = 0.5000$	$p = 0.9924$	$p = 0.9272$
4	4	2	**11**	$p = 0.9738$	$p = 0.9982$	$p = 0.1660$
5	**7**	**7**	1	$p = 0.5000$	$p = 0.5000$	$p = 0.9990$
6	**12**	1	6	$p = 0.1794$	$p = 0.9999$	$p = 0.9157$
7	3	**10**	2	$p = 0.9806$	$p = 0.1508$	$p = 0.9951$
8	**14**	1	0	$p = 0.0009$***	$p = 0.9990$	$p = 0.9998$
10	7	0	**12**	$p = 0.8206$	$p = 1.0000$	$p = 0.1794$
11	**12**	6	1	$p = 0.1794$	$p = 0.9157$	$p = 0.9999$
12	**12**	1	5	$p = 0.1193$	$p = 0.9998$	$p = 0.9505$
13	**10**	5	3	$p = 0.4068$	$p = 0.9505$	$p = 0.9952$
14	**22**	6	0	$p = 0.0022$**	$p = 0.9977$	$p = 1.0000$
15	**13**	6	0	$p = 0.0843$*	$p = 0.9157$	$p = 1.0000$
16	**15**	8	5	$p = 0.4251$	$p = 0.9812$	$p = 0.9993$
17	1	**17**	1	$p = 0.9999$	$p = 0.0006$***	$p = 0.9999$
18	**13**	3	3	$p = 0.0843$*	$p = 0.9970$	$p = 0.9970$
19	**12**	6	1	$p = 0.1794$	$p = 0.9157$	$p = 0.9999$
Maintenance	**128**	40	32	$p = 5.031 * 10^{-05}$***	$p = 1.0000$	$p = 1.0000$
Understanding	**68**	46	22	$p = 0.5000$	$p = 0.9999$	$p = 1.0000$
Overall	**196**	86	54	$p = 0.0013$**	$p = 1.0000$	$p = 1.0000$

Sig codes: $\alpha = 0.1$*, $\alpha = 0.01$**, $\alpha = 0.001$***

task, followed by the smell number referencing Section 4. The *Artifacts* column references the artifacts in Table 2, and the *Refactoring* column indicates the type of refactoring used to remove the smell (refactorings are defined in Section 6). The *User Goal* indicates how the participant was asked to judge their preference between the dirty and clean pipes (i.e., the user goal). For illustration, Task 19 with artifact set 1 is shown in Fig. 4; the dirty pipe is A and the clean pipe is B.

The results of the study are summarized in Table 4. For each task (identified by number in the *Task* column), all tasks pertaining to maintenance, all tasks pertaining to understanding, and all tasks aggregated, we show the number of participants who preferred the *clean* pipe, *dirty* pipe, or *neither* pipe in the *All Results* column. We test for significance using a 1-sample test of given proportions to determine if the response represents a majority using the null hypothesis $H_0 : \pi_{response} \leq 0.5$ and alternate hypothesis $H_a : \pi_{response} > 0.5$ for the clean (π_c), dirty (π_d), and neither responses (π_n).[5] The p-values are reported for each of these tests.

Overall, a significant majority preferred the clean pipes at $\alpha = 0.01$. For those tasks dealing with maintenance, a significant majority preferred the clean pipes at $\alpha = 0.001$. However, for the tasks dealing with understanding (i.e., the user goal of *to understand* or *for others to understand*), a majority preferred the clean pipes, but it was not significant. Breaking it down by task, in 13 of the 18 tasks, a majority of participants preferred the clean pipe to the dirty pipe, and six tasks had a significant majority at $\alpha = 0.10$ or lower. Of the remaining five tasks, two indicate that a majority of participants expressed no preference, two showed that a majority of participants preferred the dirty pipe, and one showed a tie between the clean and dirty selections. We now group the tasks by smell and discuss the findings.

Duplicate module smells make a pipe harder to understand and perhaps also harder to maintain (Tasks 1, 3, and 19). For all tasks that used a consolidation refactoring, a majority of participants preferred the refactored pipe, but the majority was only significant for Task 1, which asked about understanding. This may be due to the size difference between pipes used for Task 1, where the dirty pipe had eight modules and the clean pipe had three (Fig. 4 depicts the pipes used in this task).

Abstraction on strings eases maintenance, but not understanding (Tasks 2, 15, 17, and 18). For string abstractions and Smell 4: Duplicate Strings, participants recognize the benefits for maintenance, but find the hard-coded values easier to understand. In the three tasks that asked about maintenance, a significant majority preferred the clean pipe. For the one task that asked about understandability (Task 17), a significant majority preferred the dirty pipe.

Preferences for abstraction on modules seems to depend on size and goal (Tasks 7, 14, and 16). With Smell 6: Isomorphic Paths, for the goal of maintenance (Tasks 14 and 16) a majority preferred the clean pipe, and for the goal of understandability (Task 7), a majority preferred the dirty pipe. In only one instance (Task 14), that with the longest paths for replacement, did a significant majority of participants prefer the refactored pipe. Participants seemed to find value when the abstraction removed modules from view, providing a greater reduction in size.

Laziness smells are never preferred (Tasks 6 and 8). Two tasks involve laziness smells, one asked about understandability and the other about maintainability. In both tasks, a majority of participants preferred the clean pipe, but the majority was significant for only one task (Task 8). This shows that laziness smells may impact understandability and maintenance, at least for these or similar tasks.

5. As opposed to a χ^2 test of equal proportions in which the proportions would be compared to 0.33, the test we used is more conservative as it compares the proportion for each proportion to the pooled proportion of the other two selections.

TABLE 5
RQ2 Tasks and Results Breakdown

Task	Smell	Artifacts	Refactoring	Treatment	All Results Correct	Incorrect	Total	$H_0 : \pi_c \leq 0.5$ $H_a : \pi_c > 0.5$
20	Redundancy: 6	15	Abstraction	clean	7	1	8	$p = 0.0385$*
				dirty	6	0	6	$p = 0.0206$*
21	Redundancy: 5 Laziness: 3	16	Consolidation, Reduction	clean	5	1	6	$p = 0.1103$
				dirty	8	2	10	$p = 0.0569$*
				all clean	12	2	14	$p = 0.0081$*
				all dirty	14	2	16	$p = 0.0029$*
				Overall	26	4	30	$p = 6.302 * 10^{-05}$***

Sig codes: $\alpha = 0.1$*, $\alpha = 0.01$**, $\alpha = 0.001$***

Invalid sources have little impact on user preferences (Tasks 4, 10, and 12). In two of the three tasks that involved the invalid sources smell, a majority of participants showed no preference between the pipes. For Task 12, a majority preferred the clean pipe for maintainability, but it was not significant. For this type of environmental smell, participants are generally indecisive. The free-form responses indicate that because the structures are basically the same between a pipe with or without this smell, the pipes are equally easy to understand or maintain. This may indicate a misunderstanding of the impact of invalid data sources (e.g., harder to debug, an unnecessary number of fields).

Deprecated modules have little impact or are not well understood (Tasks 5, 11, and 13). In all tasks involving a deprecated module, the results were not significant in the context of the user goal, though a majority preferred the clean pipe in two of the three tasks. It would seem that when it comes to maintenance, the participants do not mind this smell, which may indicate a misunderstanding about the impact of the smell (i.e., broken or failing modules).

5.2.2 RQ2: Are Pipes with Smells Less Understandable?

In the tasks for this second experiment, the programmer was presented with a clean or dirty pipe and asked to select the pipe's output. The selection was indicated using a multiple-choice question in which the output was described in English prose. The tasks are described in Table 5. The *Task* column provides a quick reference for the tasks by number. As with Table 3, the *Smell* describes the type of smell involved with the task, the *Artifacts* column references the artifacts in Table 2, and the *Refactoring* column indicates the type of refactoring used to remove the smell. The *Treatment* column breaks each task into two treatment levels; participants could respond to the clean or dirty treatment level, but not both.

The results for these tasks, per treatment, are also shown in Table 5. The tasks were scored as being *correct* or *incorrect*, and these numbers are shown for each of the tasks in the *All Results* column. A one-sample test of given proportions ($H_0 : \pi_{correct} \leq 0.5$) for three of the tasks and treatments (excluding Task 21, clean) revealed that a significant proportion of participants selected the correct answer with $\alpha = 0.1$. Performing a two-sample test of equal proportions with the clean versus dirty pipes within each task does not reject the null hypothesis $H_0 : \pi_{correct,clean} = \pi_{correct,dirty}$ ($p = 1.0000$ for each task). Overall a significant majority selected the correct result at $\alpha = 0.001$. This observation held when considering the dirty pipes (significant at $\alpha = 0.01$) or the clean pipes (significant at $\alpha = 0.01$). Thus, the presence of the smells used in these tasks did not seem to impact pipe understandability.

5.3 Study Summary

Returning to the research questions, for *RQ1*, it was found that clean pipes are preferred over dirty pipes ($\alpha = 0.01$). For maintenance tasks, a significant majority preferred the clean pipe ($\alpha = 0.001$), yet for understanding tasks, a nonsignificant majority preferred the clean pipe.

The string abstraction was preferred for maintainability, but the lack of abstraction was preferred for understandability. Additionally, a majority of participants preferred the pipes that lack duplicate module smells. For the environmental smells, however, participants generally showed no preference when it came to invalid sources, but for lazy smells, the refactored pipe was almost always preferred. For deprecated modules, while a majority generally preferred the pipe that lacked the smells, it was not a significant majority. For pipes with isomorphic paths, the size of the abstraction impacts user preferences.

For *RQ2*, participants were able to select the correct output of the pipes with high accuracy, regardless of the presence or absence of smells. This indicates that the smells might not actually impede understandability, but here, the number of tasks is small so these results may not generalize to a larger study. Other confounding factors may also have impacted the results. Since the multiple-choice answers were generated by the researchers, bias may have been introduced to make the correct solution more obvious, or possibly the absence of strict time constraints or monetary incentive caused participants to put in more effort. Further study is needed to fully understand if smells impact the understandability of pipes. We discuss these and other threats in detail in Section 8. Additionally, further study is needed to determine that smelly pipes are more or less maintainable, and a possible future direction would be to

give people maintenance tasks on Pipes and evaluate whether or not they were successful with such tasks. This is left for future work.

6 Refactorings

To address the most prevalent code smells, we have defined a set of semantic preserving pipe refactorings. Following Opdyke's definition for behavior preservation in terms of the set of outputs resulting from the same set of inputs [11], we define two pipes as being semantically equivalent if the set of unique items that reaches each pipe's final output module are the same, ignoring duplicate items and items' order (correctness proof sketches are available [26]). Ultimately, it is the set of items that matters for the output of this domain, as the items can easily be reordered by inserting a *sort* module prior to output.

Since a pipe is a graph, we build on the concepts of graph transformation to specify these refactorings. A pipe refactoring is then a transformation $refactor : P_{before} \rightarrow P_{after}$, where P_{before} is the refactoring precondition represented by a smell defined in Section 4, and P_{after} is the refactoring postcondition. Each smell identifies wires, modules, and paths that are used by the refactoring transformation, and these are identified as the *Parameters*, or *Params*. Each refactoring transformation is decomposed into a set of more basic transformations, which is identified as *Transf.* These transformations take the form of $[predicate] \rightarrow action$, where the predicate may or may not be present, and the action is a commonly performed operation (set, create, add, remove, copy, append, prepend) on pipe components (paths, modules, wires, and fields). Multiple actions will be separated by a comma, such as $\rightarrow$ set $w.dest = w_j.dest$, set $w.fld = w_j.fld$, remove w_j, which sets the destination of wire w to the destination of w_j, and then removes w_j from the pipe, as is done as part of Refactoring 2.2: Lazy Module.

For example, in Refactoring 1: *Clean Up Module*, the precondition is a disjunction, with one clause requiring an instance of Smell 1.1: *Empty Field*. This smell identifies a field f in a module m where the field is empty; f and m are passed in as parameters. The transformation step requires no predicate, but simply states, remove f from $m.\mathcal{F}$. Here, the postcondition is that f is no longer contained in $m.\mathcal{F}$.

As another example, which involves a predicate in the transformation step, we look to Refactoring 2.1: *Disconnected, Dangling, or Swaying*. Here, we consider the precondition of Smell 2.1: *Cannot reach output*, which can identify, for example, a (nonoutput) module that has an incoming wire but no outgoing wire, and therefore is dangling. The precondition only identifies the module m, so the parameters passed in only contain m. When removing the module m from the $Pipe$, all wires connected to m must also be removed. So, in the refactoring transformation, we look for all wires ($\forall w \in \mathcal{W}$) such that $in\_wire(m, w)$, $out\_wire(m.w)$, or $fld\_wire(m, w)$ evaluate to true, indicating a connected wire to m. If such a wire exists, and therefore the predicate evaluates to true, then the action, remove w, is performed. As a final step of the transformation, m is removed. The postcondition states that $m \notin Pipe$, since no matter what, this transformation will remove the smelly module m. Next, we provide the refactoring definitions.

6.1 Reduction

These refactorings focus on removing unnecessary fields and modules that result from duplicated or lazy components, resulting in a smaller, more simplified pipe.

Refactoring 1. Clean up module removes empty or duplicated fields within a module. While the transformation is the same for both of these cases (i.e., the problematic field is removed), the motivations are different. Removing empty fields is analogous to the "Remove Parameter" refactoring, which removes parameters that are "no longer used by the method body." This refactoring is the most commonly performed as reported in a recent survey, where 70 percent of the developers said to perform this refactoring manually [27]. For duplicate fields, if the "same code structure [exists] in more than once place," the code will be better without the duplication [9].

P_{before}	Smell 1.1: Empty Field $\vee$ (Smell 1.2: Duplicated Field $\wedge$ $gen(m)$)
Params	$Pipe$, module m, field f
Transf.	remove f from $m.\mathcal{F}$
P_{after}	$f \notin m.\mathcal{F}$

This refactoring, when applied for Smell 1.2: *Duplicate field*, is safe for generator modules because it will simply remove duplicate items from the pipe; by our definition of semantic preservation, this is acceptable.

Refactoring 2. Remove noncontributing modules remove two kinds of unnecessary modules, those that are poorly placed in the pipe (e.g., modules that do not reach the output) and those that are ineffectual (e.g., operator modules that do not contain fields). This refactoring is designed to address the "Lazy Class" code smell, which emphasizes that all code "costs money to maintain and understand," so code that "isn't doing enough should be eliminated" [9].

Case 2.1. Disconnected, Dangling, or Swaying—Modules that are isolated, do not reach the output, or are at the top of a path but do not generate any items, are unnecessary.

P_{before}	Smell 2.1: Cannot reach output $\vee$ Smell 2.5: Swaying module
Params	$Pipe$, ineffectual module m
Transf.	$\forall w \in \mathcal{W}((in\_wire(m, w) \vee out\_wire(m, w)$ $\vee fld\_wire(m, w)) \rightarrow$ remove w) remove m
P_{after}	$m \notin Pipe$

Module E in Fig. 2a meets the precondition pertaining to Smell 2.1: *Cannot reach output*. Thus, the ineffectual module m in the $Params$ would map to module E in Fig. 2a. In the transformation, all the wires leading to or from module m are removed (there are none), and then m is removed. The postcondition, then, is that m is removed from the pipe. In Fig. 2a, this postcondition is met in the transformation to Fig. 2b since E is removed.

Case 2.2. Lazy module—That does not perform any operation or performs unnecessary redirection can be removed.

P_{before}	Smell 2.2: Ineffectual path altering $\vee$ Smell 2.3: Inoperative module $\vee$ Smell 2.4: Unnecessary redirection
Params	$Pipe$, ineffectual module m
Transf.	$\forall w_j \in \mathcal{W}(out\_wire(m, w_j) \rightarrow$ (

$(\forall w \in \mathcal{W}\ ((in\_wire(m, w) \wedge w \neq w_j)$
$\rightarrow$ set $w.dest = w_j.dest$,
set $w.fld = w_j.fld$,
remove w_j))
$\wedge\ (\forall w \in \mathcal{W}\ ((fld\_wire(m, w) \wedge w \neq w_j)$
$\rightarrow$ set $w.dest = w_j.dest$,
set $w.fld = w_j.fld$,
remove w_j))))
remove m

P_{after} $m \notin Pipe$

In the transformation from Fig. 2a to Fig. 2b, modules A and B consolidate to $A + B$ and module C disappears. This is the results of two refactorings performed in sequence. The first, Refactoring 5 *Collapse Duplicate Paths*, transforms A and B into $A + B$. This creates an instance of Smell 2.2: *Ineffectual path altering*, meeting the precondition of this refactoring. That is, when A and B are consolidated, then there is one wire into, and one wire out of, module C. In this refactoring, the ineffectual module m maps to module C. In the transformation step, the input wire from m is rerouted so the destination is the same as the output wire from m. Then, the output wire from m is removed. Finally, m is removed, meeting the postcondition that $m \notin Pipe$. This is shown by the absence of module C in Fig. 2b.

Refactoring 3. Inline module removes setter modules that have only one outgoing wire, as these can be replaced with string values in the destination field without sacrificing abstraction. This refactoring is inspired in part by the "Inline Method" refactoring that will "put the method's body into the body of its callers and then remove the method" [9].

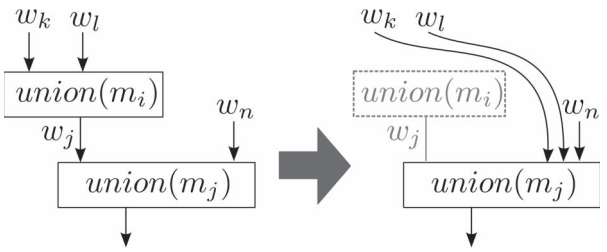
P_{before} Smell 3: Unnecessary Abstraction
Params $Pipe$, unnecessary module m, wire w_i
Transf. String $s =$ ""
append $m.\mathcal{F}$ to s,
set $(w_i.fld).value = s$,
remove w_i,
remove m
P_{after} $m, w_i \notin Pipe$

This refactoring creates a string s that contains the concatenation of all the fields in the unnecessary module m. Then, the field that receives its value from w_i, identified as $w_i.fld$, is set to s.

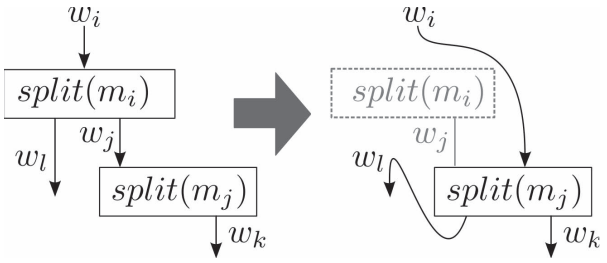
6.2 Consolidation

These refactorings aim to unify duplicated code to simplify pipe structures and reduce their sizes, a desirable pipe characteristic expressed by end users (see Section 5). These refactorings merge operator modules performing actions that could be completed with just one module and collapse duplicate paths that perform identical actions on separate lists of items, which are later merged.

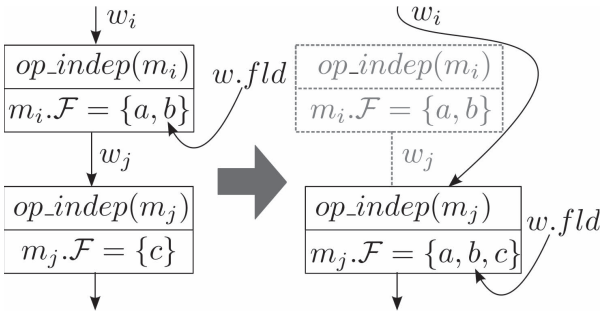
Refactoring 4. Merge Redundant Modules merges operator modules that perform the same or similar operations along the same path, or path-altering modules with the same type that are connected, hence decreasing the size and complexity of the pipe. This refactoring is motivated in part by the "Inline Class" refactoring that moves all the features of one class into another class, and then deletes it [9]. Here, m_i is being inlined, and m_j absorbs its features.



(a) Merge Redundant *union* Modules



(b) Merge Redundant *split* Modules



(c) Merge Redundant Operator Modules

Fig. 5. Merge redundant, connected modules.

Case 4.1. Connected and redundant modules can be consolidated by merging the fields, for operator modules, or reconnecting the wires, for path-altering modules.

P_{before} Smell 5.1: Consecutive redundant modules
Params $Pipe$, operators m_i, m_j, connecting wire w_j
Transf. $\forall w \in \mathcal{W}\ ((in\_wire(m_i, w) \wedge (w \neq w_j)$
$\wedge\ (union(m_i) \vee split(m_i) \vee op(m_i)))$
$\rightarrow$ set $w.dest = m_j$,
set $w.fld = \emptyset$)
$\forall w \in \mathcal{W}\ ((out\_wire(m_i, w) \wedge (w \neq w_j)$
$\wedge\ split(m_i)) \rightarrow$ set $w.src = m_j$)
$(op(m_i) \wedge !same\_fld\_values(m_i, m_j))$
$\rightarrow$ prepend $m_i.\mathcal{F}$ to $m_j.\mathcal{F}$
remove m_i, w_j
P_{after} $m_i, w_j \notin Pipe$

Fig. 5 shows connected *union* modules that are underutilized (Fig. 5a), connected *split* modules that can be consolidated (Fig. 5b), and connected operator modules that are merged (Fig. 5c). More concretely, the modules in Fig. 5a meets the precondition, Smell 5.1: *Consecutive redundant modules*, in that module m_i and m_j are both $union$ modules, and there exists a wire w_j that joins them. In the transformation, all the input wires to m_i are rerouted to the output of w_j (the input of m_j), and then m_i and w_j are removed. This meets the postcondition that $m_i, w_j \notin Pipe$.

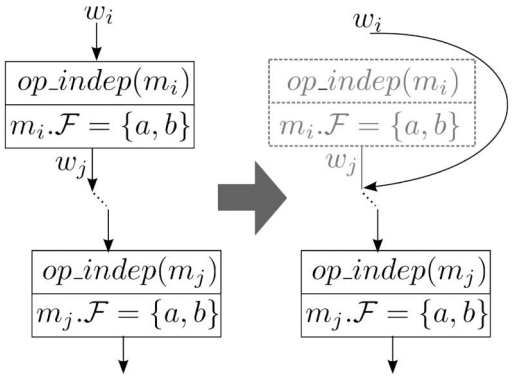


Fig. 6. Identical subsequent operator modules.

Using the example from Fig. 2a, hypothetically, if H was removed and C connected directly to J, this would meet the precondition. In that situation, m_i would map to C and m_j would map to J. In the absence of H, there would be a wire, w_j joining C to J. The transformation would identify and remove w_j, rewire all incoming wires to C down to J, and remove J. This would meet the precondition, where m_i (J) is not in the pipe.

Case 4.2. Identical, subsequent operators—That perform the same action at different locations in the pipe can be simplified.

P_{before}	Smell 5.2: Identical subsequent operators
Params	$Pipe$, operators m_i, m_j
Transf.	$\exists w_j, w_i \in W((out\_wire(m_i, w_j)$
	$\wedge\ in\_wire(m_i, w_i))$
	$\rightarrow$ set $w_i.dest = w_j.dest$,
	set $w_i.fld = w_j.fld$,
	remove w_j)
	$\forall w \in \mathcal{W}(fld\_wire(m_i, w) \rightarrow$ remove $w)$
	remove m_i
P_{after}	$m_i \notin Pipe$

An example of this refactoring in which the operator modules are not directly connected yet perform the same operation along the same path is shown in Fig. 6. Recall that in the definition of Smell 5.2: *Identical subsequent operators*, only *union* or order-independent, read-only operator modules are allowed to separate the identical modules. Otherwise, the refactoring would be unsafe.

Refactoring 5. Collapse duplicate paths—Aggregated paths can often be consolidated into a single path to simplify the pipe structure. This refactoring is motivated in part by the "Form Template Method" refactoring, which takes two methods that perform similar steps in the same order and eliminates the duplication [9]. However, in our case, instead of forming a template method in a superclass, we form a template path and collapse two similar paths into one.

Case 5.1. *Joined Generators*

P_{before}	Smells 5.3: Joined generators
Params	$Pipe$, modules m_i, m_j and wires w_i, w_j
Transf.	append $m_i.\mathcal{F}$ to $m_j.\mathcal{F}$,
	remove m_i, w_i
P_{after}	$m_i, w_i \notin Pipe$

For example, the pipe in Fig. 2a meets the precondition of this transformation, and the pipe in Fig. 2b meets the postcondition. Module m_j maps to module A in Fig. 2a, and m_i maps to B. The field in B (i.e., the URL) is added to module A, so that $|A.\mathcal{F}| = 2$. Then, B and its outgoing wire are removed. The updated A module with the added field is shown in Fig. 2b by module $A + B$. Fig. 7 also shows the refactoring, but for a pipe with joined generators that has a connected *setter* module. This is represented by the wire $w.fld$ that is rerouted to a field in m_j.

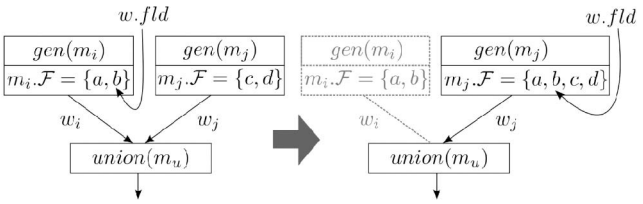


Fig. 7. Collapse duplicate paths, joined generators.

Case 5.2. Identical parallel operator

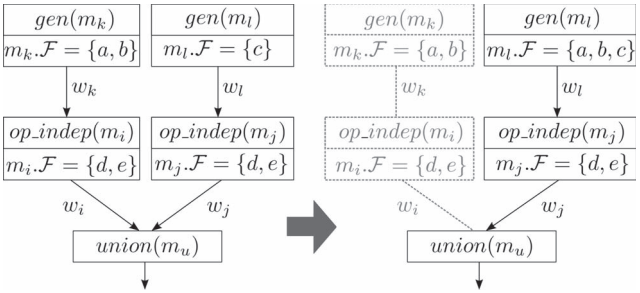
P_{before}	Smells 5.4: Identical parallel operators
Params	$Pipe$, modules m_i, m_j, m_k, m_l,
	wires w_i, w_j, w_k, w_l
Transf.	$((gen(m_k) \wedge gen(m_l))$
	$\rightarrow$ append $m_k.\mathcal{F}$ to $m_l.\mathcal{F}$,
	remove m_k, w_k)
	$((gen(m_k) \wedge union(m_l))$
	$\rightarrow$ set $w_k.dest = m_l$, set $w_k.fld = \emptyset$)
	$((union(m_k) \wedge gen(m_l))$
	$\rightarrow$ set $w_l.dest = m_k$, set $w_l.fld = \emptyset$)
	$\forall w \in \mathcal{W}((in\_wire(m_k, w) \wedge union(m_k)$
	$\wedge\ union(m_l))$
	$\rightarrow$ set $w.dest = m_l$, set $w.fld = \emptyset$)
	$((union(m_k) \wedge union(m_l))$
	$\rightarrow$ remove m_k, w_k)
	remove m_i, w_i
P_{after}	$m_i, w_i \notin Pipe$

We illustrate this refactoring in Fig. 8. Fig. 8a shows a pipe with two parallel operators and two preceding generator modules, Fig. 8b shows a pipe with two parallel operators with a union module and generator module preceding, and Fig. 8c shows a pipe with parallel operators and two preceding union modules.

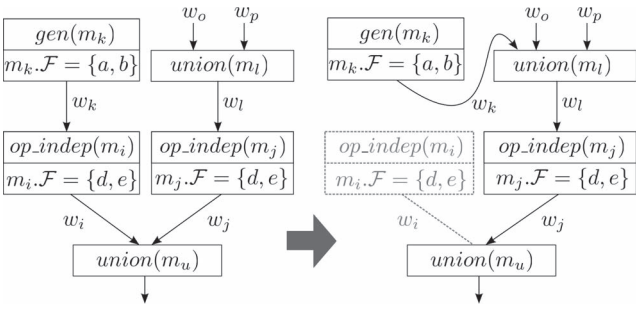
As a hypothetical example, if in Fig. 2a, modules G and I were identical $filter$ modules, then this would meet the precondition where m_i maps to G, m_j maps to I, m_k maps to D and m_l maps to F. In that instance, those two paths could be collapsed into one path by appending the fields from D into F, and then removing modules D and G. This would meet the postcondition where m_i is not in the pipe.

6.3 Abstraction

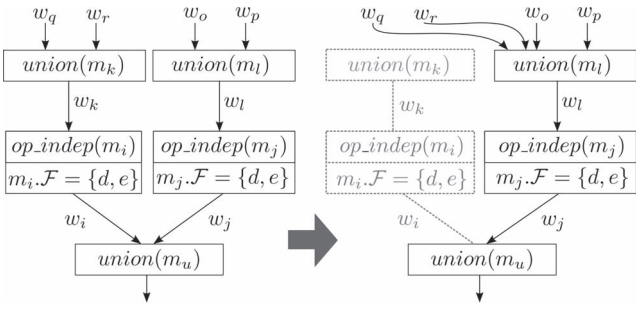
These refactorings abstract sections of the pipe that have duplicate fields or modules. They are in part inspired by the "Pull Up Method" refactoring, which aims to extract common code from subclasses into a superclass to increase maintainability [9], something for which it is hard to provide automated support. In a survey of 328 developers, over half manually perform this refactoring in practice [27]. These refactorings either create new modules that provide values to existing modules or replace existing modules.



(a) Identical Parallel Operators with *generator* and *generator*



(b) Identical Parallel Operators with *union* and *generator*



(c) Identical Parallel Operators with *union* and *union*

Fig. 8. Collapse duplicate paths, identical parallel operators.

Refactoring 6. Pull up module extracts duplicate strings into a newly created module, m. This new module provides the string values via new wires to the previous owners of the duplicated strings.

P_{before} Smell 4: Duplicate Strings
Params $Pipe$, fields f_i and f_j
Transf. add module m to $\mathcal{M}$,
set $m.type = setter.string$,
add field g to $m.\mathcal{F}$,
set $owner(g) = m$,
set $g.value = f_i.value$,
add wire w_i to $\mathcal{W}$,
set $w_i.src = m$,
set $w_i.dest = owner(f_i)$,
set $w_i.fld = f_i$,
add wire w_j to $\mathcal{W}$,
set $w_j.src = m$,
set $w_j.dest = owner(f_j)$,
set $w_j.fld = f_j$
P_{after} $m, w_i, w_j \in Pipe$
$\wedge\ g \in m.\mathcal{F}$

An example of this refactoring is shown from Fig. 2a to Fig. 2b, where module M was added to provide the string value "3" to modules H and $F + I$.

Refactoring 7. Extract local subpipe creates a subpipe that contains the modules in isomorphic paths in a pipe, and replaces those paths with the subpipe. The replacement of the path with a semantically equivalent subpipe is similar to the *Substitute Algorithm* refactoring that replaces an algorithm with one that is cleaner [9]. Here, we replace all instances of the path with a single, cleaner module. For example, in Fig. 2b, a subpipe was created to replace two paths from Fig. 2a, from D to G, and from F to I. The field values from D, F, G, and I were copied to their respective subpipes. The wire providing the field value to I was reconnected to the field from I in subpipe $F + I$.

P_{before} Smell 6: Isomorphic Paths
Params $Pipe$, isomorphic paths p and p'
Transf. percent Build subpipe
(1) create pipe $newPipe$
add module o to $newPipe.\mathcal{M}$
set $o.type = output$
copy p to $newPipe$
add wire v to $newPipe.\mathcal{W}$
set $v.src = p(last)$
set $v.dest = o$
set $v.fld = \emptyset$
(2) $\forall f \in newPipe((f.wireable = true) \rightarrow ($
add module q to $newPipe.\mathcal{M}$,
set $q.type = setter.user$,
add wire x to $newPipe.\mathcal{W}$,
set $x.src = q$,
set $x.dest = owner(f)$,
set $x.fld = f))$
(3) $\forall$ path $a \in Pipe((a = p \vee a = p') \rightarrow ($
add module r to $Pipe.\mathcal{M}$,
set $r.name = subpipe(newPipe)$
add wire t to $Pipe.\mathcal{W}$
set $t.src = r$
set $t.dest = a(last + 1)$
set $t.fld = \emptyset$
$\forall$ module $m \in a(\forall f \in m.\mathcal{F}($
$(f.wireable = true) \rightarrow ($
$(\exists w \in Pipe.\mathcal{W}((w.fld = f)$
$\rightarrow$ set $w.dest = r.q))$
$\vee$ (copy $f.value$ to $r.q.value))))$
remove $a))$
P_{after} $p, p' \notin Pipe$
$\wedge\ \exists_2 subpipe(newPipe) \in Pipe$

In part (1) of the refactoring, a new pipe is created to replicate the isomorphic path. The new modules are added and wired together. In part (2), the fields are set up so they can be set dynamically when this pipe is included as a subpipe in (3). For all fields that are wireable, adding a user-setter module allows these values to be set dynamically. In Part (3), the new pipe is added as a subpipe module (i.e., r) within $Pipe$. All field values are copied from the isomorphic paths p and p' into r, and the new subpipe module is wired into place. Then, the paths p and p' are removed from $Pipe$.

6.4 Deprecations

Outdated or broken modules and data sources can lead to unexpected pipe behavior. These refactorings either replace or remove such pipe components to increase the dependability of the pipe.

Refactoring 8. Replace deprecated modules assume that a function $replace(\mathcal{M}) \rightarrow \mathcal{M}$ exists that takes a deprecated module, m_{dep}, and returns a module or sequence of modules, M_{new}, that perform a semantically equivalent operation as m_{dep}. This refactoring is similar in spirit to previous work that uses refactorings to update references to deprecated library classes in Java programs [28]. One difference is that in our work, it is not up to the programmer to specify the mapping between the deprecated and replacement modules; this is done on behalf of the programmer. This refactoring assumes that the number of incoming and outgoing wires to and from m_{dep} and M_{new} are the same.

P_{before} Smell 7: Deprecated Module
Params $Pipe$, module m_{dep}, M_{new}
Transf. add M_{new} to $Pipe$
$\forall w_i \in \mathcal{W}(in\_wire(m_{dep}, w_i)$
$\rightarrow$ set $w_i.dest = M_{new}(first))$
$\forall w_j \in \mathcal{W}(out\_wire(m_{dep}, w_j)$
$\rightarrow$ set $w_j.src = M_{new}(last))$
remove m_{dep}
P_{after} $m_{dep} \notin Pipe \wedge M_{new} \in Pipe$

For example, the $babelfish$ module has been deprecated by the Yahoo! Pipes environment (Section 7.1). If there exists a module in a pipe, where $m.name = babelfish$, then this meets the precondition and $m_{dep} = m$. In the refactoring, m_{dep} is removed from the pipe and replaced with M_{new}, a module or list of modules that replicates the behavior of m_{dep}, but is supported by the environment. This, then, meets the postcondition where $m_{dep} \notin Pipe$ and $M_{new} \in Pipe$.

Refactoring 9. Remove deprecated sources remove all sources that refer to invalid external data sources to reduce the bloat and remove a common cause of pipe failures. The ability to perform this refactoring is intrinsic to the mashup domain as the external sources can be easily checked for validity.

P_{before} Smell 8: Invalid Sources
Params $Pipe$, field f referring to an invalid es
Transf. $\forall m \in \mathcal{M}((m = owner(f)$
$\rightarrow$ remove f from $m)$
P_{after} $f \notin Pipe$

For example, if a generator module accesses a broken data source es, then it does not contribute anything to the pipe (similar to E in Fig. 2a, which is disconnected). Removing the field f that points to es rids the pipe of the deprecated source, and meets the postcondition that $f \notin Pipe$.

6.5 Population-Based Standardizations

These refactorings exploit the availability of a large public repository of pipe-like mashups to standardize the programming practices across the community and facilitate reuse.

Refactoring 10. Normalize order of operations reorder the order-independent, read-only operator modules to match the ordering prescribed by the population. These are modules that can be reordered without impacting the ultimate pipe output. The goal is to increase the understandability of the pipes by enforcing a de facto, standard ordering on the operators that has been defined by the population.

P_{before} Smell 9: Nonconforming module orderings
Params $Pipe$, nonconforming path p,
prescribed path $ppres$
Transf. add $ppres$ to $Pipe$
(1) $\forall w_i \in \mathcal{W}(in\_wire(p(first), w_i)$
$\rightarrow$ set $w_i.dest = ppres(first))$
$\forall w_j \in \mathcal{W}(out\_wire(p(last), w_j)$
$\rightarrow$ set $w_j.src = ppres(last))$
(2) $\forall m \in p(\text{copy } m.\mathcal{F} \text{ to } ppres(m).\mathcal{F})$
remove p
P_{after} $ppres$ in place of p

In part (1), the prescribed path $ppres$ is wired into the pipe to replace p. In part (2), all the fields from p are copied to their respective modules in $ppres$, and then p is deleted.

For example, if $p = [filter, sort, filter]$ is a nonconforming path, and $ppres = [sort, filter, filter]$ is a conforming path, this refactoring will extract p from the pipe, insert $ppres$, and parameterize all fields of all modules in $ppres$ according to the fields in p.

Refactoring 11. Extract global subpipe—A generalization of Refactoring 7 to operate across a population of pipes, broadening the space on which the pattern identification occurs. As the search space for candidate paths increases, the cost of this refactoring needs to be controlled by constraining either the size of the path or the population being considered. This refactoring assumes that a function $getSubPipe(Path) \rightarrow Pipe$ exists that takes an isomorphic path and returns a global pipe that can replace it (each subpipe is built like those in Refactoring 7, lines (1-2)).

P_{before} Smell 10: Global Isomorphic Paths
Params isomorphic $Paths$
Transf. Start with parts (3) in Refactoring 7.
Replace the first three lines of Part (3) with:
$\forall$ path $a \in Pipe((a = p) \rightarrow ($
add module r to $Pipe.\mathcal{M}$,
set $r.name = getSubPipe(a)$
P_{after} $p \notin Pipes$
$\wedge \exists_1 subpipe(newPipe) \in Pipe$

As an example, consider the pipe in Fig. 2a. If isomorphic paths, D to G, and F to I are passed in as parameters, this means there exists a pipe in the community that can be included as a subpipe. Each of these paths is then replaced by a subpipe module returned by the $getSubPipe()$ method. In Fig. 2b, this postcondition is met when D and G are replaced by subpipe $D + G$, and likewise with F and I being replaced by $F + I$.

7 Empirical Study

To measure the frequency of smells (reported in Section 4) and effectiveness of refactorings presented in Section 6, we obtained a sample of pipes by scraping 10,360 pipes from Yahoo!'s public repository, and then filtered based on size.

We constrained our search queries to pipes containing at least one of the 20 most popular data sources (as reported in January 2010), independently of the pipe structure. The sample average size is 8.5 modules per pipe, and we only retain those pipes with at least four modules (the minimal number necessary to create a pipe with multiple paths to the output using two generators, one union, and one output). This resulted in the final sample of 8,051 pipes.

This section presents the adaptation of the implementation of the refactorings to fit the Yahoo! Pipes language, the infrastructure we built to perform the study, and the results.

7.1 Refactoring for Yahoo! Pipes

This section describes the additional refactoring constraints and adaptations we performed to fit the Yahoo! Pipes language. We later discuss the impact of these changes in Section 7.3.

Refactoring 3: Inline module. The $urlbuilder$ module required additional processing to insert separator symbols when assembling a url string from its fields (e.g., base url, parameters). That is, "http://" was appended to the beginning of the base URL, the paths were separated by "/", and so forth to create a valid URL from the $urlbuilder$ fields.

Refactoring 4: Merge redundant modules and *refactoring 5: Collapse duplicate paths*—These refactorings require $op(m_i)$ to accommodate multiple fields, so it was only implemented for $sort$, $filter$, $regex$, and $rename$. For operators with nonwireable fields, matching constraints were added requiring the nonwireable fields to match prior to merging. For example, on the *filter* module, to allow merging, both modules need to match on the inclusion/exclusion criteria, which is set in this language using two fields. The first defines inclusion or exclusion on the filter and can be set to *permit* or *block*. The second criteria defines if the filter criteria should be connected with an *and* operation (denoted *all*), or an *or* operation (denoted *any*). It is only safe to combine two filters that both specify *permit any* or *block any*; it is not safe to combine filters that perform *permit all* or *block all*, as these could include or exclude extra items, respectively. Last, path-altering modules in Yahoo! Pipes have a bounded number of potential wires. We added preconditions to respect those bounds (limits of five incoming wires for $union$ and two outgoing wires for $split$).

Refactoring 8: Replace deprecated modules—Yahoo! Pipes provides a list of deprecated modules and some suggestions on how to replace them. Our implementation supports replacement of the following deprecated modules: $foreach$, $foreachannotate$, $contentanalysis$, and $babelfish$.

Refactoring 9: Remove deprecated sources —This refactoring is applied to generator and string-setter modules, but not to user-setter modules because the url can be changed at runtime.

Refactoring 10: Normalize order of operations and *Refactoring 11: Extract global subpipe* —We generate $PPres$ and $PGPaths$ by considering the pipes cloned more than 10 times (~10 percent of the pipes in the population). For *Refactoring 10* we identified paths of size two to five, containing read-only and order-independent modules and for *Refactoring 11* we identified paths of length three or more that appear in multiple pipes within the subset.

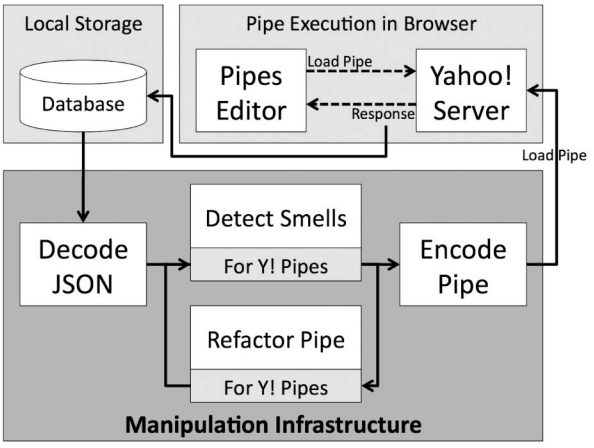


Fig. 9. Study infrastructure.

7.2 Study Infrastructure

To perform this study, we had to obtain a pipe representation, analyze it to detect smells, and refactor it. Yahoo!, however, does not provide an API to perform any of those actions outside their proprietary Pipes Editor. We built an infrastructure that allows us to perform these tasks efficiently (each analysis and transformation takes less than a second except for the smells that require a query to external sources) on thousands of pipes to detect the prevalence of smells and assess the refactorings effectiveness in reducing the smells. This infrastructure is depicted in Fig. 9.

By executing searches on the Yahoo! Pipes repository, we obtained ids for those pipes that met our selection criteria. For each id, we then sent a *load pipe* request to Yahoo!'s servers; the response contained a JSON [29] representation of the Pipe in the POST data. We stored the results in a database and built a manipulation infrastructure that can decode, detect smells, refactor, and re-encode the pipes so they can be re-executed on Yahoo!'s servers. Our prototype sends the encoding of the refactored pipes to Yahoo!'s servers so the newly updated version can be loaded in the Pipes Editor. The GUI component of the prototype is not mature enough for release as it was intended for research purposes only. The manipulation infrastructure, however, has been made available.[6] This infrastructure contains analyzers for all smells, can perform all refactorings[7] subject to the language constraints just described, and supports the full grammar of Yahoo! Pipes. It takes four parameters. The first gives the option to run the smell detection with or without the refactoring, and the second determines the output format, a command line description, DOT format of pipe, or both. The third parameter is the refactoring to perform, followed by the fourth parameter, an identifier for the pipe to analyze.

6. http://www.cs.iastate.edu/~kstolee/refmash.html.

7. The refactorings provided through the tool infrastructure are intended to operate independently of Yahoo!, so some smells and refactorings are not available in this version of the infrastructure. The omitted analyses and transformations involve 1) an analysis of community artifacts, or 2) are dependent on either the Yahoo! server or the repository, both of which can change over time. The Global Isomorphic Paths smell, Extract Global Subpipe refactoring, and Extract Local Subpipe refactoring have such dependencies. As an additional note, the Invalid Sources smell and the Nonconforming Operator Ordering smells utilize the invalid sources and operator orderings that were determined in February 2010.

TABLE 6
Smell Frequency out of 8,051 Pipes

#	Smell Name	Frequency	Impact
1	Noisy Module	28%	The presence of this smell makes a pipe harder to maintain (Task 6)
2	Unnecessary Module	13%	The presence of this smell makes a pipe harder to understand (Task 8***)
3	Unnecessary Abstraction	12%	The presence of this smell may make a pipe harder to understand (Task 21)
4	Duplicate Strings	32%	The presence of this smell makes a pipe harder to maintain (Tasks 2**, 15*, 18*), but easier to understand (Task 17)
5	Duplicate Modules	23%	The presence of this smell makes a pipe harder to understand (Tasks 1**, 3, 21) and perhaps also harder to maintain (Task 19)
6	Isomorphic Paths	7%	The presence of this smell makes a pipe easier to understand (Tasks 7, 20), but for larger abstractions makes a pipe harder to maintain (Tasks 14**, 16)
7	Deprecated Module	18%	For pipes with similar complexities, the absence of this smell is preferred (Tasks 11, 13), but if removing the smell increases the pipe complexity, user preferences are unclear (Task 5)
8	Invalid Sources	14%	The presence of this smell may make a pipe harder to maintain (Task 12), but generally this smell has little impact on user preferences (Tasks 4, 10)
9	Non-conforming Module Orderings	19%	The presence of this smell makes a pipe harder to understand (Task 8***)
10	Global Isomorphic Paths	6%	The presence of this smell makes a pipe easier to understand (Task 7)
	Total	**81%**	Sig codes: $\alpha = 0.1$*, $\alpha = 0.01$**, $\alpha = 0.001$***

As part of the infrastructure (not part of the tool), we also implemented a wrapper that repeatedly runs the smell detector and the refactorings that address those smells until no further smell reduction can be obtained. This helps us explore how refactorings may interact when applied in sequences. The wrapper operates with an outer loop that runs until no smells can be removed, a middle loop that iterates on all the current smells in the pipe, and an inner loop that applies refactorings targeting the current smells.

Algorithm 1. Greedy application of refactorings.
Require: Pipe $\mathcal{PG} = (\mathcal{M}, \mathcal{W}, \mathcal{F}, owner)$
$Map <Smell, Refactoring> smellRefMap$
Ensure: returns $\mathcal{PG}'$, a pipe with minimal smells
$Set <Refactoring> ref$
$Set <Smell> currentSmells, previousSmells$
$\mathcal{PG}' = \mathcal{PG}$
$currentSmells = detectSmells(\mathcal{PG}')$
$previousSmells = \emptyset$
while $previousSmells$!= $currentSmells$ **do**
  for $s \in currentSmells$ **do**
    $ref = smellRefMap.getAll(s)$
    for $r \in ref$ **do**
      $refactor(\mathcal{PG}', r)$
    end for
  end for
  $previousSmells = currentSmells$
  $currentSmells = detectSmells(\mathcal{PG}')$
end while
return $\mathcal{PG}'$

Algorithm 1 illustrates how the wrapper operates. The outer loop ensures that the algorithm will continue until no smells can be removed. The middle loop iterates on all the current smells in the pipe, using the $smellRefMap$ to identify the refactorings that may reduce the smell. The inner loop applies all the relevant refactoring. This algorithm allows us to take advantage of refactorings that open the doors for others to be applied, and for those refactorings that target different aspects of a smell and can be applied simultaneously. Our approach to this, however, is iterative. A more effective approach to reducing code smells may be to leverage search-based refactoring techniques that use simulated annealing or genetic algorithms to optimize refactoring sequences toward a goal (e.g., [30], [31]). Evaluating the impact of such optimizations is left for future work.

7.3 Results of Artifact Analysis

Using the study infrastructure, we analyzed the sample of pipes to determine the prevalence of smells and the effectiveness of the refactorings at removing those smells.

7.3.1 Prevalence and Impact of Smells

The frequency of occurrence for each smell defined in Section 4 is summarized in Table 6, alongside the impact of those smells as uncovered in the user study (Section 5). The *Smell* column lists all the smells, followed by the *Frequency* column, which indicates the percentage of pipes that contain the particular smell. For example, as shown in the first row, Smell 1: *Noisy Module* appears in 28 percent of the 8,051 pipes. The redundancy smells have the highest frequencies; duplicated strings exist in 32 percent of the pipes and duplicate modules appear in 23 percent of the pipes. Overall, we identified at least one smell in 81 percent of the pipes and on average, each pipe contains approximately eight instances of two different smells. The *Impact* column links each smell and its frequency to the general findings from the user study (Section 5). The significance markers indicate which tasks had a significant majority of users that preferred the clean pipe, as reported in Table 4.

We observe that the four most common smells are generally not preferred by users (Smells 1: *Noisy module*, 4: *Duplicate strings*, 5: *Duplicate modules*, and 9: *Nonconforming module orderings*), as well as the seventh most common (Smell 2: *Unnecessary module*). In the study, these smells were removed using reduction, consolidation, or population-based standardization refactorings. Other less common smells seem to be preferred for maintenance but not for understanding (Smells 6: *Isomorphic paths* and 10: *Global isomorphic paths*) or the participants showed no strong preference (Smells 3: *Unnecessary abstraction* and 8: *Invalid sources*). The preferences for one smell depended on the complexity of the refactored pipe, where the smelly pipe was preferred if the refactored version added too much

TABLE 7
Refactoring Effectiveness in Reducing Smells in Pipes

Refactorings		Smells									
		Laziness			Redundancy			Environmental		Population-Based	
		Noisy Module	Unnecessary Module	Unnecessary Abstraction	Duplicate Strings	Duplicate Modules	Isomorphic Paths	Deprecated Module	Invalid Source	Module Ordering	Global Paths
Smells Per Pipe		5.27	2.03	1.81	12.52	5.10	5.64	1.54	2.57	1.00	1.25
Reduction	Clean Up Module	-18.4%									
	Non-Contrib. Module		-100%								
	Inline Module	-11.3%		-100%	-10.2%						
Consol.	Merge Modules					-17.2%					
	Duplicate Paths					-72.0%					
Abstract.	Pull Up Module	-12.7%		-47.4%	-100%						
	Local Subpipe				-11.7%		-100%				-23.0%
Deprecat.	Deprecated Module							-100%			-7.1%
	Deprecated Source		+24.7%						-99.2%		
Populat.	Module Ordering									-100%	
	Global Subpipe										-100%
Greedy Approach		-42.7%	-100%	-100.0%	-100%	-89.7%	-100%	-100%	-99.2%	-100%	-100%

complexity (Smell 7: *Deprecated module*). These smells are addressed using abstraction or environmental refactorings.

This relatively high frequency of smells (Table 6), paired with the evidence that programmers generally prefer pipes that lack smells, implies a need for refactorings. Next, we evaluate the effectiveness of the refactoring transformations at removing smells in the repository.

7.3.2 Refactoring Effectiveness

For each smell, Table 6 presents the frequency of occurrence in the repository and Table 7 presents the number of smell instances (smelliness) per pipe in the *smells per pipe* row. Each subsequent row in Table 7 shows the change in smelliness after applying each individual refactoring. For example, each pipe affected by Smell 5: *Duplicate modules* contains an average of 5.10 smelly modules. After applying Refactoring 5: *CollapseDuplicate paths*, each affected pipe has an average of 1.43 smelly modules, a reduction of 72 percent. The table presents results if the change in smells per pipe was greater than 5 percent.

Seven of the refactorings applied individually are able to completely remove certain smells from the pipes: Refactoring 2: *Remove noncontributing module* eliminates Smell 2: *Unnecessary module*, Refactoring 3: *Inline module* eliminates Smell 3: *Unnecessary abstraction*, Refactoring 6: *Pull up module* eliminates Smell 4: *Duplicate strings*, Refactoring 7: *Extract local subpipe* eliminates Smell 6: *Isomorphic paths*, Refactoring 8: *Replace deprecated modules* eliminates Smell 7: *Deprecated module*, Refactoring 10: *Normalize module ordering* eliminates Smell 9: *Nonconforming module orderings*, and Refactoring 11: *Extract global subpipe* eliminates Smell 10: *Global isomorphic path*. Refactoring 9: *Remove deprecated sources* is almost as effective, eliminating over 99 percent of Smell 8: *Invalid sources* instances.

We note that some refactorings cause changes that open the door for other refactorings to be performed. For example, Refactoring 9: *Remove deprecated sources* not only eliminates 99 percent of Smell 8: *Invalid source*, but it also increases the presence of Smell 2: *Unnecessary module* by 25 percent (removing deprecated sources can lead to a module with no fields, fitting Smell 2.3). This creates an opportunity for Refactoring 2: *Remove noncontributing module*. Generally, the study participants showed no preference toward to absence of Smell 8: *Invalid source*, so in practice this situation might not occur. However, should a noncontributing module appear, users showed a preference toward its absence (which would involve Refactoring 2: *Remove noncontributing module*). Other refactorings may have small individual impact, but can be applied in combination with others to target different aspects of a smell to have a greater overall effect. For example, three refactorings have a valuable impact on Smell 1: *Noisy module*, with a maximum individual reduction of 18 percent, but a collective reduction closer to 43 percent. Since the study participants preferred the absence of this smell, the

collective impact could be important to give the greatest impact to the end-user programmers.

We explore the effect of applying a sequence of refactorings utilizing the greedy Algorithm 1 to take advantage of the compounding effect of multiple refactorings. The results, shown in the last row of Table 7, indicate that seven smells are completely eliminated in all the affected pipes. However, even when applying the refactorings greedily, not all the smells can be eliminated. Smell 1: *Noisy module* is not eliminated because the implementation of Refactoring 1: *Clean up module* only targets the generator and setter modules. Smell 5: *Duplicate modules* is not eliminated because of the implementation limitations of Refactoring 4: *Merge redundant modules*; there are many consecutive union modules that have reached maximum capacity on their input wires. Smell 8: *Invalid source* is not eliminated because Refactoring 9: *Remove deprecated sources* does not remove sources within user-setter modules.

7.4 Summary of Artifact Studies

Overall, before applying the refactorings, 6,503 of the 8,051 pipes had at least one smell, which represents nearly 81 percent of the population. After applying all the refactorings in the greedy approach, only 1,323 of the pipes have smells, representing 16 percent of the pipes. This means that the refactorings were able to completely eliminate the smells in nearly 80 percent of the pipes that had smells to begin with. On average, the number of smell instances per pipe was reduced from eight to one through the proposed refactorings.

8 Threats to Validity

Resulting from the user study and artifact study, there are several threats to validity worth mentioning.

Conclusion. In the user study, in some cases there are very few participants, particularly with the tasks associated with RQ2, which leads to low statistical power. To compensate for those, we chose conservative statistical tests so not to overestimate the significance of the findings.

Internal. The user study results might be subject to history effects due to the study context, Mechanical Turk. The tasks were completed on several different days, so the study circumstances were different. Additionally, some of the tasks used the same artifacts but asked about different user goal; for those users who performed both tasks associated with an artifact, maturation effects may have been present. Self-selection is another effect to note, as all participants selected our tasks from Mechanical Turk and volunteered to participate in the study.

Construct. In the user study, we presented the user with two pipes, one that had a smell and another that did not. In some cases, it might not have been the presence of the smell, but rather the level of the smelliness that caused the effect. For the isomorphic paths smells, we were able to tease this out a bit by showing that just one of the tasks—that in which the smell had the greatest impact on the pipe—were the results significant. For some of the significant results, it might be the intensity of the smell that influenced the outcome, rather than the presence of the smell. Conversely, for those results in which significance was not observed, it might be that the particular instance of the smell was too subtle to detect an effect.

Additionally, although the presence of smells was shown to negatively impact programmers' preferences, we did not assess the proposed refactorings in the hands of end users to understand whether and how they are adopted in practice. Rather, we based their preferences on their self-reported evaluation of the artifacts, with respect to understandability and maintainability, which may not be representative of actual understandability or maintainability.

Related to the artifact study, the refactorings presented in Section 6 are guaranteed to generate pipes that produce the same set of unique items (reflecting Opdyke's definition [11]; proof sketches are available [26]). However, the proposed refactorings may cause a pipe to return data items in a different order, if an order is not made explicit in the pipe. For example, a refactoring may change the order in which data are fetched and then integrated through a union unless a sort module follows. Still, a refactoring tool could address such issues by enforcing an order through the addition of an extra sorting module or by simply warning the programmer about potential side effects prior to the transformation.

External. The participants in the user study were people on Mechanical Turk; these populations may not be representative of those who would program and use a refactoring tool. However, since nearly 50 percent of the participants in the study reported holding degrees in computer science, and all the participants were required to pass a qualification test to participate, we feel this may mitigate the effect of interaction between selection and treatment.

Additionally, the setting in which the Mechanical Turk participants completed the study might not be representative of a normal development environment, and we do not know if that had an influence on the results. The users were not able to play with the pipes to view their behavior, rather, they were provided with a static view and asked to judge their preferences, which might not be a realistic situation. We did allowed them to view the pipe in full resolution, but the interaction of setting and treatment remains.

In the artifact study, the 8,000 artifacts studied represent only a sample of the population, and are subject to sampling bias. To mitigate the risk, we used a selection criteria unrelated to the program structures, which still could have biased the applicability of the refactorings.

9 Related Work

Two areas of related work are most relevant, on end-user programming and web mashups, and on refactoring.

9.1 End-User Programming and Web Mashups

Mashup development environments target a wide range of users and provide various levels of development support [2]. Environments oriented toward more proficient developers often require knowledge of scripting languages (e.g., Plagger requires perl programming [32]), but a recent trend has been toward environments and languages that allow programmers to work at higher levels of abstraction. These environments often wrap common mashup tasks (e.g., fetching data in known formats, aggregating, filtering) into

preconfigured modules, trading flexibility and control for lower adoption barriers. The environments' languages provide visual mashup representations, with the pipe structure/flow representation being common among commercial mashup development environments (e.g., Yahoo! Pipes [3], Apatar [5], DERI Pipes [6], Feed Rinse [7], IBM Mashup Center [8], JackBe [33], and xFruits [34]).

Another interesting trend among the commercial tools is the emergence of communities around these environments to provide end-user programmer support, either as a forum, wiki, or as a repository of mashups to be shared with other programmers [3], [5], [6], [8], [33]. Previous studies of these communities have looked at the social structure evidenced by the communication on messageboards [4], or the uniqueness of artifacts [35], but not at the quality of or deficiencies present in the artifacts.

Researchers have also sought to support mashup developers in the composition of mashup programs. Some work focuses on creating and evolving mashups [12], [13], [14], [15], [16], [17], [36], and other work aims to extend existing mashup environments with domain-specific support [37], [38] or techniques to more easily integrate heterogeneous data sources [16], [39], [40]. In mashup composition with existing languages, some researchers aim to assist users by suggesting components or larger program pieces while the mashup is being created based on mined patterns [12], [13], [14], [15] and/or tags [12]. Other work has created new mashup languages to perform common mashup operations, such as importing, merging, sorting, and reporting data [16], [17]. In an effort to lower the learning curve, some researchers have proposed domain-specific extensions to existing mashup languages that use terminology that might be familiar to more novice programmers [37]. While the level of support for mashup creation is increasing, the level of support for facilitating maintenance, understanding, and robustness of mashups is just starting to be noticed [18].

9.2 Refactoring

Although no refactoring support exists yet for mashup tools, the body of work on refactoring is extensive [22]. The concept of refactoring was first introduced as a systematic way to restructure code to facilitate software evolution and maintenance [10], [11]. Since then, the scope and type of refactorings has grown considerably. For example, refactoring has been used to improve code design [9] and to make code more reusable and maintainable by introducing type parameters [41] and specific design patterns [42]. Tools have also been created to update references to deprecated library classes [28] and parallelize sequential code by introducing calls to libraries that support writing concurrent programs [43]. At a slightly higher level of abstraction, refactoring of program structures has also been used to facilitate feature decomposition and feature-based changes during program evolution [44]. Our work pulls some inspiration from these techniques (specifically, [9], [28]), but also introduces some novel refactorings based on environmental limitations and community patterns [1].

Within the context of model-driven software development, refactoring has been applied at the design level, mostly through UML transformations to, for example, support program evolution [45] or facilitate the transformation of different types of UML diagrams [46]. Although not exclusively [47], it is among such model refactorings that we often see the use of graph transformations as a mechanism to explicitly define the preconditions, postconditions, and transformation steps [48]. Graph transformations have been used to facilitate refactoring of software models [46], [49] and to detect dependencies among different refactorings [50]. Further, graph rewriting has been shown to be suitable for expressing refactoring transformations and proving the preservation of certain program properties [47]. We have adopted a similar graph-based approach to make explicit the smells and the refactoring preconditions, postconditions, and transformations we introduce.

Field studies have sought to uncover the benefits of refactoring [27], the frequency with which programmers invoke automated refactoring tools [51], and how the automated tools are used in practice [27], [52]. From surveying 328 programmers at Microsoft, it was found that refactoring typically reduces post-release defects, and that most refactorings are performed manually. This fits with the results of a study of 41 developers who use the Eclipse IDE, where it was found that only 25 percent use the automated refactoring tools [51]. Another field study found that programmers typically perform small refactorings and use the tools when they are aware of the refactorings available [52]. An observation that is consistent across the studies that interviewed programmers is that developers are willing to perform refactorings that change the program behavior [27], [52], which is relevant to our discussion in Section 6.

The evaluations of refactoring techniques have focused on languages utilized by professional software developers, though recent refactoring and smell detection research has started to target the spreadsheet domain [23], [24]. In studies that target professional developers, a typical course of evaluation is to implement the refactorings in a tool and evaluate it on a set of programs, measuring time to complete a refactoring [28], [41], [43], changes in program size [28], [42], or accuracy compared to the manually performed refactorings [41], [43]. We follow a similar approach, taking advantage of a public repository of mashup programs to perform a study on a large population of mashups to determine the prevalence of the smells and the effectiveness of the refactorings in addressing those smells. In addition, we also perform a study to examine the impact of smells in 20 pipes on 61 end-user programmers' mashup preference and understanding. Unlike recent refactoring evaluations involving human studies [27], [51], [52], we focus on whether or not users preferred the *outcome* of a refactoring as opposed to studying their current refactoring behaviors, which would not be feasible. It would have been premature to perform a user study without first understanding programmer preferences with respect to smells.

10 Discussion

In this section, we discuss the generalizability of our approach to another web mashup language and explore extensions to other end-user programming domains.

Many emerging environments are enabling end users to create increasingly sophisticated mashups. Our study, however, focuses on just one of those environments, Yahoo! Pipes. This environment was selected to maximize the potential impact of the findings (given the popularity Yahoo! Pipes), and because of the availability of a rich public repository to support a large study on smell detection and refactorings. Still it remains to be explored whether the smells and refactorings will be relevant in other environments. In this section, we briefly investigate the applicability of the refactorings defined in this work to another mashup language, DERI Pipes, and speculate on how other end-user programming languages and environments could leverage these mashup refactorings to suit their individual characteristics.

10.1 DERI Pipes Analysis

To assess the generalizability of these refactorings as defined, we performed a manual inspection and analysis of the pipes available in the newer DERI Pipes repository. This language uses modules and wires to define the data and control flow of mashup programs, similarly to Yahoo! Pipes. Of the 139 published DERI pipes (Aug 2010), 77 meet the size selection criteria used for our Yahoo! Pipes study, with an average of 1.4 total smells per pipe. In spite of the smaller pool size, we find that five of the eight smells we searched for (population-based smells were not considered as their manual analysis was deemed too expensive) are present in these pipes.

We note, however, that particular DERI language constructs and constraints will require further tailoring of our infrastructure. For example, since DERI's generator modules do not support multiple fields, they cannot be merged, so *Smell 5: Duplicate module*, which affects 30 percent of the pipes, cannot be used as implemented. Still in these pipes we observe that three out of the five smells can be successfully detected and refactored. *Smell 6: Isomorphic paths* impacts 10 percent of the pipes, and can be eliminated using *Refactoring 7: Extract local subpipe*. *Smell 8: Invalid source* impacts 9 percent of the pipes, and can be eliminated using *Refactoring 9: Remove deprecated sources*. Smell 1: Noisy module impacts 8 percent of the pipes, and Smell 2: Unnecessary module impacts 6 percent of the pipes. Each of these smells can be eliminated using the refactorings described in Section 6. Additionally, 6 percent of the pipes contain unnecessary modules that can be removed with *Refactoring 2: Remove NonContributing modules*.

10.2 Extensions to Other Domains

In this section, we discuss how the refactorings defined in Section 6 can apply or extend to other end-user programming domains, specifically spreadsheets, web macros, and educational programming environments. Additionally, we outline future opportunities for refactorings in these other domains.

10.2.1 Spreadsheets

One of the most popular end-user programming domains is spreadsheets, and it has recently been the target of smell detection and refactoring research efforts [23], [24]. The refactorings target smells such as hard-coded constraints, duplicated expressions, and unnecessary complexity [23], [24], which are analogous to smells in our work, specifically Smell 4: *Duplicate strings*, Smell 5: *Duplicate modules*, and Smell 2: *Unnecessary module*, respectively. Another area that could be explored would be population-based smells and refactorings in which the community of spreadsheets could be analyzed for patterns, such as a common setup for an accounting spreadsheet. Similar spreadsheets that deviate from the common structures could be marked as smelly because it might be harder for others to understand.

10.2.2 Web Macros

Web macros are programs that automate tasks performed on the web, like accessing and moving data among a set of spreadsheets, forms, and websites. Several tools facilitate the easy creation of web macros by recording a user's interactions and automatically creating a program that will replay the actions [20], [53]. Recent research in this domain has aimed to increase the dependability of the macros by adding assertions [20]. Similar to the web mashup domain, web macros also have a dependence on external data sources. The environmental Smell 8: *Invalid Sources* and Refactoring 9: *Remove deprecated sources* related to deprecated data sources apply directly to this domain. The spirit behind Smell 2: *Unnecessary module* can be used to identify dead or unreachable code in a web macro script, possibly resulting from the use of conditionals.

A category of smells that we have not fully explored targets access-related issues, such as access-denied server responses or when a user is logged out of a system. This is smelly because it can change the behavior of the pipe without warning if the user logs out of a web application or if another user tries to copy and reuse the program. A refactoring that addresses this smell could be useful in the mashup and macro domains, or in any domain with such a dependence on external web data sources. In the case of web macros, the refactoring could introduce some error handling to log the user back into the application. Additionally, performance-based refactorings could identify tasks that are independent and could be performed in parallel, such as accessing and interacting with two different websites.

Another refactoring could target a current concern of web macro users. In a recent study that interviewed information workers about their needs in automation and usage of a web macros tool, CoScripter [53], it was reported that some users stopped using CoScripter because of privacy concerns in sharing the scripts. We could define a smell that detects when private information is not being hidden, and a refactoring could abstract out personal information into a private database—which CoScripter supports—to avoid sharing this personal information. This refactoring would lend itself toward a new category of refactorings for the purpose of privacy.

10.2.3 Educational Programming Environments

A new area that has yet to be explored for refactoring is educational programming environments, such as Scratch [54], Alice [55], or Kodu [56]. For Scratch and Alice, the programming languages and abstractions are similar to those used in object-oriented languages like Java, and so refactoring for those languages could follow from the

traditional refactoring literature [9]. Kodu is an event-driven language where the objects are programmed individually like autonomous agents [56], so a different approach may be needed. In Kodu, the programs are composed of many *when - do* rules to define object behavior. Each object's current behavior is defined by a page, and the behavior can change by switching to a different page. Detection of unreachable pages would be straightforward, and analogous to the Smell 2: Unnecessary module. Detecting duplicate rules or pages would be similar to Smell 5: *Duplicate modules* or Smell 6: *Isomorphic paths*. Population-based smells could identify common organization schemes for the program statements. For example, rules related to movement, earning points, and interacting with other objects could each be grouped to form a community-driven programming standard.

It is clear that there are many opportunities for refactoring in end-user languages, beyond the mashup work presented here and the recent refactoring in spreadsheets. We are just beginning to understand how refactoring can address the needs of end-user programmers, and much more exploration is needed.

11 Conclusion

End users are developing and sharing mashups in increasing numbers. However, a popular kind of mashup created by end users, pipe-like mashups, has many smells such as being bloated with unnecessary modules, accessing broken data sources, using atypical constructs, or requiring changes in multiple places even for minor updates because of the lack of abstraction. We have identified the most prevalent smells in a population of 8,051 pipes, and have found that end users prefer pipes that lack smells, particularly for maintenance. Inspired by how refactoring can benefit professional developers, we have also defined refactorings that effectively target and remove the smells. The refactorings include some adapted from more traditional programming domains (e.g., the abstraction refactorings), some that are intrinsic to the mashup domain (e.g., remove deprecated sources), and others that are novel to this work and can be generalized to other domains in which there is a public repository of community code (e.g., the population-based refactorings). The assessment of these refactorings revealed that they can reduce the frequency of smelly pipes in the population from 81 to 16 percent and reduce the average smell instances per pipe by almost 90 percent. We have also outlined how many of the smells and refactorings detected for Yahoo! Pipes programs can be found in another mashup language, DERI Pipes, and also in programs from other end-user programming domains, specifically spreadsheets, web macros, and educational games. Given these results, the next steps are to study these refactorings in the hands of end users to better understand how they can be leveraged most effectively and to continue to grow the corpus of refactorings that apply to end-user programming languages.

Acknowledgments

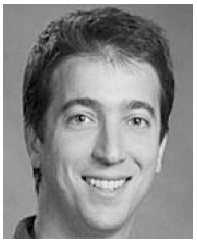
This work was supported in part by US National Science Foundation (NSF) Graduate Research Fellowship CFDA#47.076, NSF Award #0915526, and AFOSR Award #9550-10-1-0406. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies. The authors would like to thank the EUSES consortium members for their feedback on previous versions of this work. A previous version of this work appeared in the International Conference on Software Engineering [1].

References

[1] K.T. Stolee and S. Elbaum, "Refactoring Pipe-Like Mashups for End-User Programmers," *Proc. 33rd Int'l Conf. Software Eng. (ICSE '11)*, pp. 81-90, 2011.
[2] J. Wong and J. Hong, "What Do We 'Mashup' When We Make Mashups?" *Proc. Fourth Int'l Workshop End-User Software Eng. (WEUSE)*, pp. 35-39, 2008.
[3] "Yahoo! Pipes," http://pipes.yahoo.com/, July 2009.
[4] M.C. Jones and E.F. Churchill, "Conversations in Developer Communities: A Preliminary Analysis of the Yahoo! Pipes Community," *Proc. Fourth Int'l Conf. Communities and Technologies (C&T '09)*, pp. 195-204, 2009.
[5] "Apatar," http://www.apatar.com/, Aug. 2009.
[6] "DERI Pipes," http://pipes.deri.org/, Aug. 2009.
[7] "Feed Rinse," http://feedrinse.com/, Jan. 2010.
[8] "IBM Mashup Center," http://www.ibm.com/software/info/mashup-center/, Aug. 2009.
[9] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[10] W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring," *ACM Trans. Software Eng. Methodology*, vol. 2, pp. 228-269, July 1993.
[11] W. Opdyke, "Refactoring Object-Oriented Frameworks," PhD dissertation, Univ. of Illinois at Urbana-Champaign, 1992.
[12] A.V. Riabov, E. Boillet, M.D. Feblowitz, Z. Liu, and A. Ranganathan, "Wishful Search: Interactive Composition of Data Mashups," *Proc. 17th Int'l Conf. World Wide Web (WWW '08)*, pp. 775-784, 2008.
[13] F. Daniel, C. Rodriguez, S.R. Chowdhury, H.R.M. Nezhad, and F. Casati, "Discovery and Reuse of Composition Knowledge for Assisted Mashup Development," *Proc. 21st Int'l Conf. Companion on World Wide Web (WWW '12 Companion)*, pp. 493-494, 2012.
[14] H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin, "Mashup Advisor: A Recommendation Tool for Mashup Development," *Proc. IEEE Int'l Conf. Web Services (ICWS '08)*, pp. 337-344, 2008.
[15] O. Greenshpan, T. Milo, and N. Polyzotis, "Autocompletion for Mashups," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 538-549, Aug. 2009.
[16] D.E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, "Damia: Data Mashups for Intranet Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08)*, pp. 1171-1182, 2008.
[17] J. Wong and J.I. Hong, "Making Mashups with Marmite: Towards End-User Programming for the Web," *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI '07)*, pp. 1435-1444, 2007.
[18] L. Grammel, C. Treude, and M.-A. Storey, "Mashup Environments in Software Engineering," *Proc. First Workshop Web 2.0 for Software Eng. (Web2SE '10)*, pp. 24-25, 2010.
[19] M.M. Burnett, C.R. Cook, O. Pendse, G. Rothermel, J. Summet, and C.S. Wallace, "End-User Software Engineering with Assertions in the Spreadsheet Paradigm," *Proc. 25th Int'l Conf. Software Eng. (ICSE)*, pp. 93-105, 2003.
[20] A. Koesnandar, S.G. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K.T. Stolee, "Using Assertions to Help End-User Programmers Create Dependable Web Macros," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (SIGSOFT FSE)*, pp. 124-134, 2008.
[21] C. Scaffidi, B.A. Myers, and M. Shaw, "Topes: Reusable Abstractions for Validating Data," *Proc. 30th Int'l Conf. Software Eng. (ICSE)*, pp. 1-10, 2008.
[22] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126-139, Feb. 2004.
[23] S. Badame and D. Dig, "Refactoring Meets Spreadsheet Formulas," *Proc. Int'l Conf. for Software Maintenance*, 2012.

[24] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and Visualizing Inter-Worksheet Smells in Spreadsheets," *Proc. Int'l Conf. Software Eng.*, 2012.
[25] "Amazon Mechanical Turk Command Line Tool Reference," http://docs.amazonwebservices.com/AWSMturkCLT/2008-08-02/, Jan. 2010.
[26] K.T. Stolee, "Analysis and Transformation of Pipe-Like Web Mashups for End User Programmers," master's thesis, Univ. of Nebraska-Lincoln, June 2010.
[27] M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," *Proc. 20th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE '12)*, 2012.
[28] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring Support for Class Library Migration," *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 265-279, 2005.
[29] "JSON," http://www.json.org/, Aug. 2009.
[30] A. Ouni, M. Kessentini, H. Sahraoui, and M. Hamdi, "Search-Based Refactoring: Towards Semantics Preservation," *Proc. IEEE 28th Int'l Conf. Software Maintenance (ICSM)*, pp. 347-356, 2012.
[31] M. O'Keeffe and M.í Cinnéide, "Search-Based Refactoring for Software Maintenance," *J. Systems Software*, vol. 81, no. 4, pp. 502-516, Apr. 2008.
[32] "Plagger," http://plagger.org/trac, Aug. 2009.
[33] "JackBe," http://www.jackbe.com/, Sept. 2012.
[34] "xFruits," http://www.xfruits.com/, Aug. 2009.
[35] K.T. Stolee, S. Elbaum, and A. Sarma, "End-User Programmers and Their Communities: An Artifact-Based Analysis," *Proc. Int'l Symp. Empirical Software Eng. and Measurement (ESEM '11)*, pp. 147-156, 2011.
[36] S.K. Kuttal, A. Sarma, A. Swearngin, and G. Rothermel, "Versioning for Mashups: An Exploratory Study," *Proc. Third Int'l Conf. End-User Development (IS-EUD '11)*, pp. 25-41, 2011.
[37] I. Muhammad, D. Florian, C. Fabio, and M. Maurizio, "Reseval Mash: A Mashup Tool that Speaks the Language of the User," *Proc. ACM Ann. Conf. Extended Abstracts on Human Factors in Computing Systems Extended Abstracts (CHI EA '12)*, pp. 1949-1954, 2012.
[38] S. Soi and M. Baez, "Domain-Specific Mashups: From All to All You Need," *Proc. 10th Int'l Conf. Current Trends in Web Eng. (ICWE '10)*, pp. 384-395, 2010.
[39] A. Bozzon, M. Brambilla, M. Imran, F. Daniel, and F. Casati, "On Development Practices for End Users," *Search Computing*, S. Ceri and M. Brambilla, eds., pp. 192-200, Springer, 2011.
[40] R.J. Ennals and M.N. Garofalakis, "Mashmaker: Mashups for the Masses," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07)*, pp. 1116-1118, 2007.
[41] A. Kiezun, M.D. Ernst, F. Tip, and R.M. Fuhrer, "Refactoring for Parameterizing Java Classes," *Proc. 29th Int'l Conf. Software Eng. (ICSE)*, pp. 437-446, 2007.
[42] H. Kegel and F. Steimann, "Systematically Refactoring Inheritance to Delegation in Java," *Proc. 30th Int'l Conf. Software Eng. (ICSE)*, pp. 431-440, 2008.
[43] D. Dig, J. Marrero, and M.D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," *Proc. 31st Int'l Conf. Software Eng. (ICSE)*, pp. 397-407, 2009.
[44] J. Liu, D.S. Batory, and C. Lengauer, "Feature Oriented Refactoring of Legacy Applications," *Proc. 28th Int'l Conf. Software Eng. (ICSE)*, pp. 112-121, 2006.
[45] G. Sunyé, D. Pollet, Y. Le Traon, and J. Jézéquel, "Refactoring UML Models," *Proc. Fourth Int'l Conf. the Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pp. 134-148, 2001.
[46] G. Taentzer, D. Müller, and T. Mens, "Specifying Domain-Specific Refactorings for Andromda Based on Graph Transformation," *Proc. Third Int'l Symp. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pp. 104-119, 2007.
[47] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens, "Formalizing Refactorings with Graph Transformations," *J. Software Maintenance and Evolution*, vol. 17, no. 4, pp. 247-276, 2005.
[48] L. Baresi and R. Heckel, "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective," *Proc. First Int'l Conf. Graph Transformation*, pp. 402-429, 2002.
[49] C. Köhler, H. Lewin, and G. Taentzer, "Ensuring Containment Constraints in Graph-Based Model Transformation Approaches," *Proc. Int'l Workshop Graph Transformations and Visual Modeling Techniques (GT-VMT)*, 2007.
[50] T. Mens, G. Taentzer, and O. Runge, "Analysing Refactoring Dependencies Using Graph Transformation," *Software and Systems Modeling*, vol. 6, no. 3, pp. 269-285, 2007.
[51] G.C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76-83, July/Aug. 2006.
[52] M. Vakilian, N. Chen, S. Negara, B.A. Rajkumar, B.P. Bailey, and R.E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," *Proc. Int'l Conf. Software Eng. (ICSE '12)*, pp. 233-243, 2012.
[53] G. Leshed, E.M. Haber, T. Matthews, and T. Lau, "Coscripter: Automating & Sharing How-to Knowledge in the Enterprise," *Proc. 26th Ann. SIGCHI Conf. Human Factors in Computing Systems (CHI '08)*, pp. 1719-1728, 2008.
[54] "Scratch," http://scratch.mit.edu/, Feb. 2011.
[55] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-D Tool for Introductory Programming Concepts," *Proc. Northeastern Conf. J. Computing in Small Colleges (CCSC '00)*, pp. 107-116, 2000.
[56] K.T. Stolee and T. Fristoe, "Expressing Computer Science Concepts through Kodu Game Lab," *Proc. 42nd ACM Technical Symp. Computer Science Education (SIGCSE '11)*, pp. 99-104, 2011.

Kathryn T. Stolee received the BS and MS degrees in computer science, and the PhD degree in computer science in 2013, all from the University of Nebraska-Lincoln. She is the Harpole-Pentair assistant professor in the Department of Computer Science and the Department of Electrical and Computer Engineering at Iowa State University. Her research uses program analysis to develop tools and techniques with the goal of making software easier to build, maintain, and understand. She is a member of the IEEE.



Sebastian Elbaum received the systems engineering degree from the Universidad Catolica de Cordoba, Argentina, and the PhD degree in computer science from the University of Idaho. He is a professor at the University of Nebraska-Lincoln. His research aims to improve software dependability through testing, monitoring, and analysis. He received the US National Science Foundation (NSF) Career Award, an IBM Innovation award, and two ACM SigSoft Distinguished Paper awards. He was the program chair for the International Symposium of Software Testing and Analysis, program cochair for the Symposium of Empirical Software Engineering and Measurement, coeditor for the *Information and Software Technology Journal*, and he is an associate editor of the *ACM Transactions on Software Engineering and Methodology*. He is a cofounder of the EUSES Consortium to support end-user programmers and the E2 Software Engineering Group at UNL. He is a member of the IEEE.