9-10-2009

Refactoring Pipe-like Mashups for End-User Programmers

Kathryn T. Stolee
University of Nebraska-Lincoln, kstolee@cse.unl.edu

Sebastian Elbaum
University of Nebraska-Lincoln, selbaum@virginia.edu

Refactoring Pipe-like Mashups for End-User Programmers

Kathryn T. Stolee, Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska – Lincoln
Lincoln, NE, U.S.A.
{kstolee, elbaum}@cse.unl.edu

ABSTRACT

Mashups are becoming increasingly popular as end users are able to easily access, manipulate, and compose data from many web sources. We have observed, however, that mashups tend to suffer from deficiencies that propagate as mashups are reused. 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 end-user programmers and observe that users generally prefer mashups without smells. We then introduce refactorings targeting those smells, reducing the complexity of the mashup programs, increasing their abstraction, updating broken data sources and dated components, and standardizing their structures to fit the community development patterns. Our assessment of a large sample of mashups shows that smells are present in 81% of them and that the proposed refactorings can reduce the number of smelly mashups to 16%, illustrating the potential of refactoring to support the thousands of end users programming mashups.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, reengineering*

General Terms

Languages, Human Factors, Experimentation

Keywords

End user software engineering, web mashups, refactoring

1. INTRODUCTION

A mashup is a program that manipulates and composes existing data sources or functionality to create a new piece of data or service that can be plugged into a web page 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 [26].

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Mashups have become extremely popular as software development environments incorporate visual programming languages and built-in functionality to quickly access and manipulate data. For example, in Yahoo! Pipes [20], one of the most popular mashup development environments, over 90,000 end users have created mashups like the one in Figure 1 since 2007 [11]. To program these mashups, users drag and drop predefined modules (e.g., a fetch module to retrieve data from a web source, a filter module to select a subset of the retrieved data), configure the modules by setting their fields (e.g., web sites from which to fetch the data, expressions to specify a filter criterion), and connect the modules to 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 [1], DERI Pipes [5], IBM Mashup Center [10]).

In spite of the increasing power and popularity of mashups, we have observed that they tend to suffer from common deficiencies, such as being unnecessarily complex, using inappropriate or dated modules or sources of information, assembling non-standard patterns, and duplicating values and functionality. For example, of the 8,051 pipe-like mashups we examined, approximately 23% had redundant modules, 32% had the same string hard-coded in multiple places, and 14% used sources of data that were not working as specified anymore. In total, 81% of the pipes had at least one type of deficiency. A detailed examination of those pipes and a study to assess the impact of those deficiencies on end-user programmers revealed that deficient pipes are more susceptible to failure, less likely to be adopted, and harder to understand by end users.

Even more concerning, however, is how these deficiencies propagate across a community where reuse is extremely common. For example, to date, over 87,000 pipes have been committed to Yahoo!'s public repository, and from the sample of pipes we studied, 66% had been cloned for reuse an average of 17 times 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 address such smells by performing semantic preserving transformations on their programs to remove smells, a process called refactoring [7, 9, 19].

Through this work we investigate how to bring the benefits of software engineering to end users programming mashups. We do this by focusing on automated smell identification and refactoring. While refactoring has received considerable attention in the context of professional programmers (e.g., [17]), this is the first refactoring effort targeting end users. Further, although adapting software en-

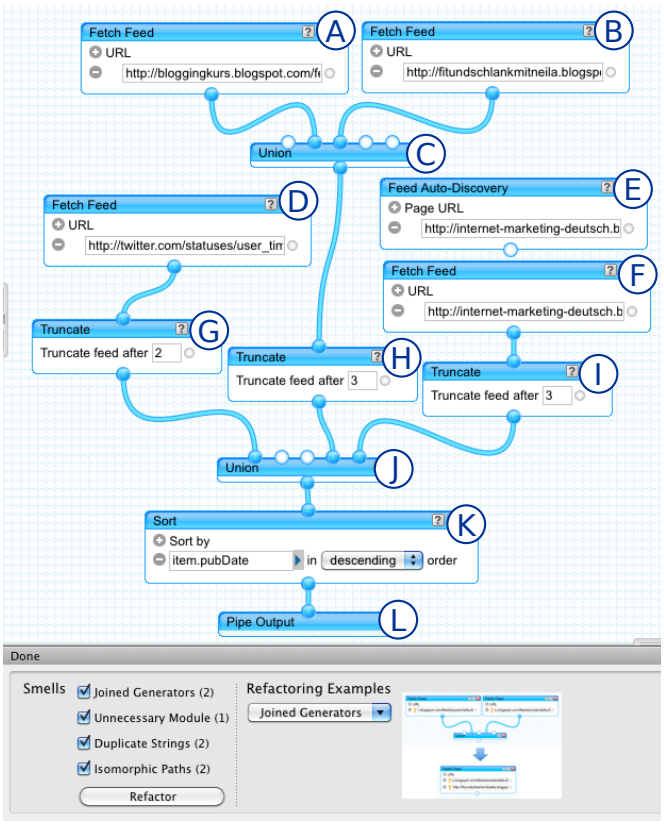

Figure 1: Original Pipe with Smells

gineering methodologies to help end users program more dependably in non-traditional language paradigms is not new in the software engineering research literature (e.g., [4, 15, 22]), this is the first effort targeting the rapidly growing pipe-like mashup domain.

This prior work notwithstanding, 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. The contributions of this work are:

- Identification of the most prevalent smells in 8,051 mashups and the assessment of the impact of those smells on 14 end-user programmers' mashup preference 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.

2. MOTIVATION

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

Figures 1 and 2 show partial screenshots 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, configuring them, and connecting them. When the user executes the mashup, it 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.

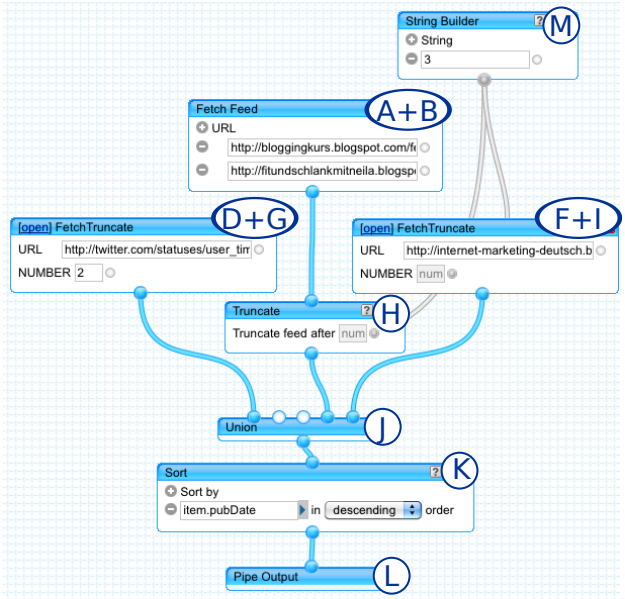

Figure 2: Completely Refactored Pipe

Figure 1 also illustrates the prototyped interface for our smell detection and refactoring toolset (grayed area). We added letter labels to serve as references to the modules to assist with the explanation. The mashup in Figure 1 was retrieved from Yahoo!'s public repository. It aggregates articles about online marketing from German websites and blogs, (e.g., the url specified in module F), and was selected for illustration due to the variety of smells it exhibits in a relatively small number of modules. The prototype indicates the smells present in the pipe, shows examples of how the smells can be refactored, and offers to perform the refactorings for the user.

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. Each box in a pipe represents 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*, *union*), and most modules contain fields that can hold hard-coded values or receive values via wire. In Figure 1, five modules retrieve data from web sources: A, B, D, E, and F, each containing one field (the web data source). These generator modules provide the inputs for the rest of the pipe modules to process. Generator 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. The outcome of generator modules A and B are aggregated through a path-altering *union* module, C, before feeding to *truncate* module H. The outgoing data from modules G, H, and I are aggregated with a *union* module, J, that feeds to a *sort* module, K, and finally to the pipe's *output* module, L.

Although this is a functional pipe, it has several smells that can be removed by refactoring, while preserving the underlying semantics. First, module E is disconnected from the rest of the pipe (no wires in or out of it) and can be trivially removed. 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. In two of the *truncate* modules, H and I, the string "3" specifies the number of data items to retain; if

the user ascertains that this value represents the same concept, then it can be abstracted into a separate module to facilitate and ensure consistency of future changes. Given that 36% of the pipes in our study were modified after being shared and 66% had been cloned at least once, making future changes easier is likely a concern for end-user programmers. Finally, two of the paths leading to module J, D to G and F to I, 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 and maintainability of the pipe.

The deficiencies and redundancies identified in Figure 1 were addressed in the fully-refactored pipe in Figure 2. Generator modules A and B were merged to yield module $A + B$, disconnected module E was removed, 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, and 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.) Merging generator modules A and B made the *union* module C ineffectual since it only one incoming wire so it was removed after the aforementioned transformations were performed. Through the refactoring 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 easier to understand.

Sections 4 – 7 proceed to define these and other smells and the complete family of refactorings in detail, and provide an assessment of the smells' frequency and impact on the end-user programmer, as well as the refactorings' effectiveness.

3. RELATED WORK

Mashup development environments target a wide range of users and provide various levels of development support [26]. Environments oriented toward more proficient developers often require knowledge of scripting languages (e.g., Plagger requires perl programming [21]), 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, 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 the most common among mashup development environments (e.g., Apatar [1], DERI Pipes [5], IBM Mashup Center [10], and Yahoo! Pipes [20]). Another interesting trend 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 [1, 5, 10, 20]. We also note that while the level of end user support for mashup creation is increasing, the level of support for facilitating maintenance, understanding, and robustness of mashups is just starting to be noticed [8].

Although no refactoring support exists yet for mashups tools, the body of work on refactoring is extensive [17]. The concept of refactoring was first introduced as a systematic way to restructure code to facilitate software evolution and maintenance [9, 19]. Since then, the scope and type of refactorings has grown considerably. For example, refactoring has been used to improve code design [7] and to make code more reusable and maintainable by introducing type parameters [14] and specific design patterns [13]. Tools have also been created to update references to deprecated library classes [2] and parallelize sequential code by introducing calls to libraries that support writing concurrent programs [6]. 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 [16].

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 [24] or facilitate the transformation of different types of UML diagrams [25]. Although not exclusively [18], 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 [3]. We have adopted a similar graph-based approach to make explicit the smells and the refactoring preconditions, postconditions, and transformations we introduce.

The evaluations of refactoring techniques have focused on languages utilized by professional software developers. In these studies, 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 [2, 6, 14], changes in program size [2, 13], or accuracy compared to the manually-performed refactorings [6, 14]. 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 on end-user programmers' mashup preference and understanding.

4. DEFINITIONS

In this section we provide definitions that will be used throughout the rest of the paper. Figure 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* $(\mathcal{F}, name, type)$*, containing a list of fields* $\mathcal{F}$ *indexed from* 1 *to* $|\mathcal{F}|$ *where* $\mathcal{F}[1]$ *is the first item in the list, a* $name$ *assigned by the Pipes programming environment (e.g.,* $fetch$ *or* $truncate$*), and a type, to be defined later.*

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.*

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

DEFINITION 4. *A pipe is a graph,* $PG = (\mathcal{M}, \mathcal{W}, \mathcal{F}, owner)$*, containing a set of modules* $\mathcal{M}$*, a set of wires* $\mathcal{W}$*, a set of fields* $\mathcal{F}$*, and a function* $owner : \mathcal{F} \rightarrow \mathcal{M}$ *assigning every field to exactly one module. The wires are constrained such that* $\forall w \in \mathcal{W}, w.src \neq w.dest$ *(no cyclic wires). Every pipe must contain exactly one module named output.*

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

As an example, Figure 1 shows a pipe with $|\mathcal{M}| = 12$, modules $A, C \in \mathcal{M}$ connected by wire $w \in \mathcal{W}$, where $w = (A, C, \varnothing)$.

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

Figure 3: Shorthand for Common Predicates Used in the Definitions, Smells, and Refactorings

Modules A and C form a path, p. where $p.length = 2, p(first) = A$, and $p(last) = C$. Module K has two non-wireable fields, (e.g., $K.\mathcal{F}[1] =$ (false,"item.pubDate")), whereas module G has one wireable field, $G.\mathcal{F}[1] =$ (true, "2").

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. A module's $m.type$ can be one of the following: 1) *generator (gen)*, if it retrieves data from external sources (e.g., an RSS feed, another pipe) and provides a list of items for other modules in the pipe to process; 2) *setter* if it only produces a value that will be wired directly into the fields of other modules; 3) *path-altering (pathAlt)* if it either joins multiple paths, as in a union, or diverts one path into multiple paths, as in a split; and 4) *operator (op)* if it manipulates a list of items.

We further subtype a *setter* module as a *string-setter* if m sets a string, or as a *user-setter* if the user may be queried to set a parameter when the pipe is executed. 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, and *read-write* ($op.rw$) if it can modify the content of list items (e.g., appending a string to each item's title). Second, o is said to be *order-independent* if the operation being performed *is not* dependent on the order of the items passed into it (e.g., reordering, renaming, or removing list items), or o is *order-dependent* 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).

5. CODE SMELLS

In this section we define the smells that were most prevalent across the sample of pipes we analyzed, and summarize the results of a first study that provides preliminary evidence about the negative impact of the defined smells on the end users' ability to understand a pipe-like mashup.

5.1 Smell Definitions

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 3), defined smells based on errors and unnecessary complexity reported in the users' newsgroups, and identified others by observing the rich sample of pipes we gathered for analysis. The pipes sample was obtained by scraping 10,362 pipes from Yahoo!'s public repository. 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 over 8 modules per pipe, and we only retain 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. Utilizing this sample, the list of candidate smells was iteratively refined as we found smells that were either not directly applicable to the domain or not common enough ($< 5\%$) to warrant their consideration.

We now define the code smells that resulted from this process. Each smell is defined in the context of a pipe represented as a graph $PG = (\mathcal{M}, \mathcal{W}, \mathcal{F}, owner)$. The frequency of occurrence is stated after each smell name (e.g., Smell 1 appears in 28% of the 8,051 pipes). Overall, we identified at least one smell in 81% of the pipes and on average, each pipe contains approximately eight instances of two different smells.

5.1.1 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" [7]. 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%):** *a module that has unnecessary fields, making a pipe harder to read and less efficient to execute. Module $m \in \mathcal{M}$ is considered noisy if:*

CASE *1.1. Empty field:*
$(gen(m) \vee setter\_str(m)) \wedge \exists f \in m.\mathcal{F} \mid f.value =$ ""

CASE *1.2. Duplicated field:*
$\exists f_i, f_j \in m.\mathcal{F} \mid same\_value(f_i, f_j, true)$

SMELL 2. **Unnecessary Module (13%):** *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:*
$\exists n \in \mathcal{M} \mid output(n) \wedge !subsequent\_mods(m, n)$

CASE *2.2. Ineffectual path altering:* $path\_alt(m) \wedge$
$\exists_1 w_i \in \mathcal{W} \mid in\_wire(m, w_i) \wedge \exists_1 w_j \in \mathcal{W} \mid out\_wire(m, w_j)$

CASE *2.3. Inoperative module:*
$!path\_alt(m) \wedge !output(m) \wedge m.\mathcal{F} = \varnothing$

CASE *2.4. Unnecessary redirection:*
$setter\_str(m) \wedge \lvert m.\mathcal{F}\rvert = 1 \wedge all\_fld\_wires(m)$

CASE *2.5. Swaying module:*
$(path\_alt(m) \vee op(m)) \wedge \nexists w \in \mathcal{W} \mid in\_wire(m, w)$

For example, in the transformation from Figure 1 to Figure 2, module E fits Case 2.1 and module C fits Case 2.2.

SMELL 3. **Unnecessary Abstraction (12%):** *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. Module $m \in \mathcal{M}$ is unnecessary if:* $setter\_str(m) \wedge \exists_1 w_i \in \mathcal{W} \mid out\_wire(m, w_i) \wedge \nexists w_j \in W \mid fld\_wire(m, w_j)$

5.1.2 Redundancy Smells

Duplicated code has been referred to as the worst smell in programs written by professionals [7]. The redundancy smells identify pipes that have duplicated strings, modules, or sequences of modules. 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%):** *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:*

$\exists f_i, f_j \in \mathcal{F} \mid owner(f_i) \neq owner(f_j) \wedge same\_value(f_i, f_j, true)$

For example, in Figure 1 the truncate modules H and I have a duplicate string "3."

SMELL 5. **Duplicate Modules (23%):** *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 operators:*
$(op\_indep(m_i) \vee path\_alt(m_i)) \wedge \exists w_j \in \mathcal{W} \mid joined\_by(m_i, m_j, w_j)$

CASE *5.2. Identical subsequent operators:*
$op\_indep(m_i) \wedge same\_fld\_values(m_i, m_j) \wedge subsequent\_mods(m_i, m_j) \wedge \forall m_k \in \mathcal{M} \mid betwn\_mods(m_k, m_i, m_j) \wedge (op\_indep(m_k) \vee union(m_k))$

CASE *5.3. Joined generators:*
$gen(m_i) \wedge conn\_to\_union(m_i, m_j)$

CASE *5.4. Identical parallel operators:*
$op\_indep(m_i) \wedge same\_fld\_values(m_i, m_j) \wedge conn\_to\_union(m_i, m_j) \wedge \exists m_k, m_l \in \mathcal{M}, \exists w_k, w_l \in \mathcal{W} \mid joined\_by(m_k, m_i, w_k) \wedge joined\_by(m_l, m_j, w_l) \wedge (gen(m_k) \vee union(m_k)) \wedge (gen(m_l) \vee union(m_l))$

Case 5.2 is shown in Figure 4 and Case 5.4 is shown in Figure 5.

SMELL 6. **Isomorphic Paths (7%):** *non-overlapping paths with the same modules performing the same operations may be missing a chance for abstraction. Two paths p and p' are isomorphic if:*

$p.length = p'.length \wedge p \cap p' = \varnothing \wedge$
$gen(p(first)) \wedge gen(p'(first)) \wedge$
$\forall m_n \in p, \forall m'_n \in p', 0 \leq n < p.length,$
$m_n.name = m'_n.name \wedge same\_number\_fields(m_n, m'_n) \wedge$
$\forall m_n \in p \mid op(m_n)$
for $k = 1 \ldots |m_n.\mathcal{F}|$
if $m_n.\mathcal{F}[k].wireable = false$ *then*
$same\_value(m_n.\mathcal{F}[k], m'_n.\mathcal{F}[k], false)$

An example is shown in Figure 1, where p consists of the path from D to G and p' consists of the path from F to I.

5.1.3 Environmental Smells

Inspired by the pervasive use of invalid and unsupported sources and modules by programs 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 that is 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%):** *a module that is no longer supported by the pipe environment. Given $Supported_{\mathcal{M}}$, a pipe presents this smell if: $\exists m \in \mathcal{M} \mid m.name \notin Supported_{\mathcal{M}}$.*

For example, four modules were deprecated in the Yahoo! Pipes environment between 2007 and 2010.

SMELL 8. **Invalid Sources (14%):** *an external data source $es \in ExternalSources$ is invalid if n consecutive attempts to retrieve data from it report errors. Given $n = 1$, a pipe presents this smell when $\exists f \in \mathcal{F}$ that refers to an invalid es.*

5.1.4 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 module patterns that have been commonly employed in highly reused pipes. Pipes using alternative module structures to implement such patterns are considered smelly since they may take more time to understand and potentially discourage reuse of those pipes across the community.

SMELL 9. **Non-conforming Module Orderings (19%):** *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 end 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 non-conforming if:*

$\forall m \in p \mid op\_indep(m) \wedge m.type = op.ro$
$\exists p' \in PPres | p \neq p' \wedge bag(p) = bag(p')$

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. We explore some values for these parameters in our study.

SMELL 10. **Global Isomorphic Paths (6%):** *building on the isomorphic path smell (Smell 6), 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:*

$\exists p \in PG, \exists p' \in PGPaths \mid p'$ *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.

5.2 End-User Programmers and Smelly Pipes

We designed two experiments to begin evaluating the impact of code smells from the perspective of the end-user programmer. The first experiment aimed to determine if the subjects prefer pipes with or without smells (RQ1), and the second aimed to determine if smelly pipes are harder to understand than clean pipes (RQ2). The experiments were split into a series of ten tasks with a random assignment of subjects to each task. In each task, we treated one pipe with a smell, providing coverage for all the smells and a variety of pipe structures. In the tasks for the first experiment, the programmer was given two pipes side by side, one with smells and the other one without, and asked to choose the preferred pipe and explain the decision. In the tasks for the second experiment, the programmer was presented with either the pipes with or without smells and asked to determine the pipe's output. (More details on the study design, tasks, and implementation can be found at [23].)

In both experiments we gauge the programmers' aptitude by asking questions about their education level and programming experience, and by requesting them to complete a pretest with eight exercises to assess their expertise with Yahoo! Pipes. The subjects were given the option to complete a tutorial on Yahoo! Pipes, and all were required to pass the pretest prior to participation, allowing us to control for subject variability. In terms of posttest measures, the first experiment evaluates user preference, the second measures correctness in identifying a pipe's output, and both measure the time to complete the task. While 50 subjects took the qualification pre-test and 34 (68%) received a passing score, a total of 22 subjects completed at least one experimental task in the study, and 14

Table 1: Impacts of Smells on End-User Programmers

Experiment 1	75 observations		
Which one of these two pipes would you prefer?	**Smelly Pipe**	**Clean Pipe**	**Same**
	24%	63%	13%

Experiment 2	16 observations	
What is the output of this pipe?	**Correct Answer**	
	Smelly Pipe	**Clean Pipe**
	67%	80%

(65%) were classified as *end users* based on their limited education in computer science (contrasted with *degreed users* who hold degrees in computer science or a related field). These end users provided a total of 91 data points across all tasks.

The results of the study are summarized in Table 1. Overall, 63% of the responses in the first experiment indicated that users preferred non-smelly pipes, while 13% were neutral about their preference. The preference for clean pipes increased to 71% for tasks involving laziness and redundancy smells where there was a general theme among all the user responses that smaller pipes with "simpler" and "cleaner" structures and fewer parameters were preferable. The preference for clean pipes jumped to 88% for the population-based smell on operator orderings. There were two tasks, however, for which subjects did not favor clean over smelly pipes. First, 56% of the end users preferred a pipe with a deprecated module rather than its clean alternative which required several additional modules. The subjects' comments seem to indicate a lack of awareness about the risks of using deprecated but still functioning modules. Second, 56% of the end users do not seem to mind the global-isomorphic path smells. When examining the comments we see that the most inexperienced programmers like to see all the pipe at once instead of abstracting functionality to another pipe that must be retrieved separately. However, more experienced programmers seemed to value abstraction but mentioned that for this particular task the abstracted functionality size did not justify the extra work.

The second experiment revealed that smelly pipes may lead to more misunderstandings about a pipe's behavior. Given a clean pipe, end-user programmers were able to correctly specify the pipe's output 80% of the time, while for smelly pipes this number dropped to 67%. In addition, the time to complete the analysis of the smelly pipe took on average 68% longer than for a clean pipe.

6. REFACTORINGS

To address the most prevalent code smells, we have devised 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 [19], 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.

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 some smells defined in Section 5, and P_{after} is the refactoring postcondition. In our specification, we have further decomposed each refactoring into a set of more basic transformations utilizing the actions performed by pipe programmers (set, add, remove, move, copy) on pipe components (modules, wires, and fields) and using visual depictions to complement the presentation of the most complex transformations.

6.1 Reduction

This category of refactorings focuses on removing unnecessary fields and modules that result from duplicated or lazy components, resulting in a pipe with fewer fields, wires, and modules.

REF 1. **Clean Up Module:** *removes empty or duplicated fields within a module. While the transformation is the same for both of these cases, 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." For duplicate fields, if the "same code structure [exists] in more than once place," the code will be better without the duplication [7].*

$\boldsymbol{P_{before}}$	*Smell 1: Noisy Module* $\wedge (gen(m) \vee setter\_str(m))$
Params	$Pipe$, *module* m, *empty or duplicated field* f
Transf.	*remove* f *from* m
$\boldsymbol{P_{after}}$	$f \notin m.\mathcal{F}$

REF 2. **Remove Non-Contributing Modules:** *removes 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 motivated by 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" [7].*

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 and can be removed.*

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

CASE *2.2.* **Lazy Module** *that does not perform any operation or performs unnecessary redirection can be removed.*

$\boldsymbol{P_{before}}$	*Smell 2.2, 2.3, 2.4: Ineffectual path altering, Inoperative module, Unnecessary redirection*
Params	$Pipe$, *ineffectual module* m
Transf.	$\exists w_j \in \mathcal{W} \mid out\_wire(m,w_j)$ $\forall w \in \mathcal{W} \mid in\_wire(m,w)$, *set* $w.dest = w_j.dest$ $\forall w \in \mathcal{W} \mid fld\_wire(m,w)$, *set* $w.fld = w_j.fld$ *remove* m, w_j
$\boldsymbol{P_{after}}$	$m, w_j \notin Pipe$

REF 3. **Push Down 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" [7].*

$\boldsymbol{P_{before}}$	*Smell 3: Unnecessary Abstraction*
Params	$Pipe$, *unnecessary module* m
Transf.	*String* $s =$ "" *for* $k = 1 \ldots \lvert m.\mathcal{F}\rvert$, *append* $m.\mathcal{F}[k]$ *to* s $\forall w \in \mathcal{W} \mid out\_wire(m,w)$ *set* $(w.fld).value = s$ *and remove* w *remove* m
$\boldsymbol{P_{after}}$	$m \notin Pipe$, $\forall w \in Pipe.\mathcal{W} \mid out\_wire(m,w)$, $(w.fld).value = s \wedge w \notin Pipe$

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 5.2). 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 that are later merged.

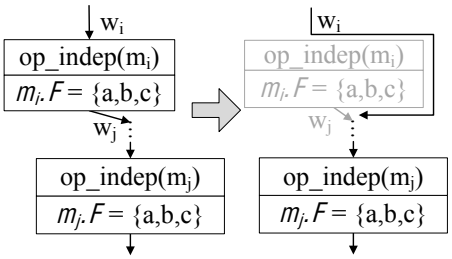

Figure 4: Merge Identical Subsequent Operator

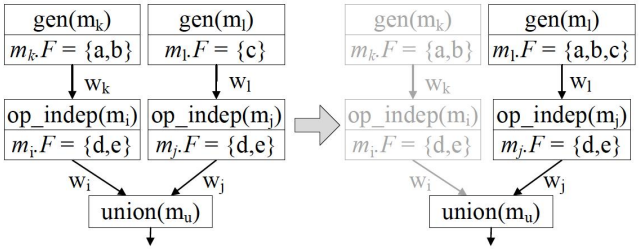

Figure 5: Collapse Duplicate Paths

REF 4. **Merge Redundant Modules:** *merges operators with the same name that perform the same or similar operations along the same path, or path-altering modules 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 [7]. Here, m_i is being inlined, and m_j absorbs all its features. See Figure 4 for an example where the operators are disconnected yet perform the same operation along the same path.*

$\boldsymbol{P_{before}}$	*Smell 5.1, 5.2: Consecutive redundant operators, Identical subsequent operators*
Params	*Pipe, operators* m_i, m_j
Transf.	$\exists w_j \in W \mid out\_wire(m_i, w_j)$
	If $union(m_i)$
	$\forall w \in \mathcal{W} \mid in\_wire(m_i, w)$,
	set $w.dest = m_j$
	If $split(m_i)$
	$\forall w \in \mathcal{W} \mid out\_wire(m_i, w)$,
	set $w.src = m_j$
	If $op(m_i)$
	$\exists w_i \in \mathcal{W} \mid in\_wire(m_i, w_i)$
	set $w_i.dest = w_j.dest$
	if $(!same\_fld\_values(m_i, m_j))$
	for $k = 1 \ldots \lvert m_j.\mathcal{F} \rvert$
	append $m_i.\mathcal{F}[k]$ *to the beginning of* $m_j.\mathcal{F}$
	remove m_i, w_j
$\boldsymbol{P_{after}}$	$m_i, w_j \notin Pipe$

REF 5. **Collapse Duplicate Paths:** *paths that are aggregated using the same union module 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 [7]. 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. An instance of this refactoring is illustrated in Figure 5.*

$\boldsymbol{P_{before}}$	*Smells 5.3, 5.4: Joined generators, Identical parallel operators*
Params	*Pipe, modules* m_i, m_j, m_k, m_l, *wires* w_i, w_j, w_k, w_l
Transf.	*if* $gen(m_k) \wedge gen(m_l)$
	for $x = 1 \ldots \lvert m_k.\mathcal{F} \rvert$
	append $m_k.\mathcal{F}[x]$ *to* $m_l.\mathcal{F}$
	remove m_k, w_k
	if $gen(m_k) \wedge union(m_l)$
	set $w_k.dest = m_l$
	if $union(m_k) \wedge gen(m_l)$
	set $w_l.dest = m_k$
	if $union(m_k) \wedge union(m_l)$
	$\forall w \in \mathcal{W} \mid in\_wire(m_k, w)$,
	set $w.dest = m_l$
	remove m_k, w_k
	remove m_i, w_i
$\boldsymbol{P_{after}}$	$m_i, m_k, w_i, w_k \notin Pipe$

6.3 Abstraction

These refactorings focus on abstracting 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 [7], something for which it is hard to provide automated support. These refactorings either create new modules that provide values to existing modules or replace existing modules.

REF 6. **Pull Up Module:** *extracts duplicate strings into a newly created module and provides the string values via wires to the previous owners of the duplicated strings.*

$\boldsymbol{P_{before}}$	*Smell 4: Duplicate Strings*
Params	*Pipe, fields with duplicate strings* f_i *and* f_j
Transf.	*add module* m *to* $Pipe.\mathcal{M} \mid setter\_str(m)$
	add field g *to* $m.\mathcal{F}$
	set $g.value = f_i.value$
	add wire w_i *to* $Pipe.\mathcal{W} \mid joined\_fld(m, f_i, w_i)$
	add wire w_j *to* $Pipe.\mathcal{W} \mid joined\_fld(m, f_j, w_j)$
$\boldsymbol{P_{after}}$	$m, w_i, w_j, g \in Pipe \mid g.value = f_i.value$
	$\wedge joined\_fld(m, f_i, w_i) \wedge joined\_fld(m, f_j, w_j)$

REF 7. **Extract Local Subpipe:** *creates a subpipe that contains the modules in the isomorphic paths in a pipe, and replaces those paths with the subpipe. For example, in Figure 2, a subpipe was created to replace two paths from Figure 1, 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.*

$\boldsymbol{P_{before}}$	*Smell 6: Isomorphic Paths*
Params	*Pipe, isomorphic paths* p *and* p'
Transf.	*% Build subpipe*
(1)	*create pipe* $newPipe$
	add module o *to* $newPipe.\mathcal{M} \mid output(o)$
	copy p *to* $newPipe$
	add wire v *to* $newPipe.\mathcal{W} \mid joined\_by(p(last), o, v)$
	$\forall f \in newPipe \mid f.wireable = true$,
	add module q *to* $newPipe.\mathcal{M} \lvert q.type = setter.user$
(2)	*add wire* x *to* $newPipe.\mathcal{W} \mid joined\_fld(q, f, x)$
	% Connect subpipe to pipe
(3)	*for (path* $a = p, p'$*)*
	add module r *to* $Pipe.\mathcal{M} \lvert r.name = subpipe(newPipe)$
	add wire t *to* $Pipe.\mathcal{W} \mid joined\_by(r, a(last + 1), t)$
	$\forall f \in \mathcal{F} \mid owner(f) \in a \wedge f.wireable = true$
	if $\exists w \in Pipe.\mathcal{W} \mid w.fld = f$, *set* $w.dest = r.q$
	if $(f.value \neq "")$, *copy* $f.value$ *to* $r.q.value$
	remove a
$\boldsymbol{P_{after}}$	p *and* $p' \notin Pipe$, $\exists_2 subpipe(newPipe) \in Pipe$

6.4 Deprecations

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

REF 8. **Replace Deprecated Modules:** *assumes 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 [2]. 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.*

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

REF 9. **Remove Deprecated Sources:** *removes all sources that refer to invalid external data sources to reduce the bloating 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.*

$\boldsymbol{P_{before}}$	*Smell 8: Invalid Sources*
Params	$Pipe$, *field* f *referring to* $es \in ExternalSources$
Transf.	*set* $m = owner(f)$ *and remove* f *from* m
$\boldsymbol{P_{after}}$	$f \notin m.\mathcal{F}$

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 in order to facilitate reuse. We are not aware of any refactoring that uses the knowledge of a crowd of programmers to determine programming standards.

REF 10. **Normalize Order of Operations:** *reorders the order-independent, read-only operator modules to match the ordering prescribed by the population. The goal of this refactoring is to increase the understandability of the pipes by enforcing a canonical ordering on the operators that has been defined by the population.*

$\boldsymbol{P_{before}}$	*Smell 9: Non-conforming module orderings*
Params	$Pipe$, *non-conforming path p, prescribed path* $ppres$
Transf.	*add path* $ppres$ *to* $Pipe$
	$\exists w_i \in \mathcal{W} \mid in\_wire(p(first), w_i)$
	set $w_i.dest = ppres(first)$
	$\exists w_j \in \mathcal{W} \mid out\_wire(p(last), w_j)$
	set $w_j.src = ppres(last)$
	$\forall m \in p$
	for $k = 1 \ldots \lvert m.\mathcal{F} \rvert$
	copy $m.\mathcal{F}[k]$ *to* $ppres(m)$
	remove p
$\boldsymbol{P_{after}}$	$ppres$ *in place of* p

REF 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. 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)).*

$\boldsymbol{P_{before}}$	*Smell 10: Global Isomorphic Paths*
Params	*isomorphic* $Paths$
Transf.	*Start at line (3) in Refactoring 7, replacing it with:*
	for$(a = p \in Paths)$, $newPipe = getSubPipe(a)$
$\boldsymbol{P_{after}}$	$\forall p \in Paths \mid p \notin Pipes, \exists_1 subpipe(newPipe) \in Pipe$

7. EMPIRICAL STUDY

To assess the effectiveness of the refactorings in removing smells, we performed an empirical study on the sample of pipes described in Section 5.1. 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, the results, and a discussion on the generalizability and validity of our findings.

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: Push Down Module. The $urlbuilder$ module required additional processing to insert separator symbols when assembling a url string from its fields (e.g., base url, parameters).

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 non-wireable fields, matching constraints were added requiring the non-wireable fields to match prior to merging. 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. The following deprecated modules are replaced: $foreach$, $foreachannotate$, $contentanalysis$, 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 run-time.

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 ($\sim$10% 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 at least length three that appear in multiple pipes within the subset.

7.2 Study Infrastructure

To perform this study we had to be able to obtain a pipe representation, analyze it to detect smells, and refactor it to reduce those smells. Yahoo!, however, does not provide an API to perform any of those actions outside their proprietary Pipes Editor. Thus, we created 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 assess the refactorings effectiveness in reducing the smells. This infrastructure is depicted in Figure 6.

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 [12] 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. This manipulation infrastructure contains analyzers for all smells, can perform all refactorings subject to the language constraints described in Section 7.1, supports the full grammar of Yahoo! Pipes, and it is also available online: *http://cse.unl.edu/~kstolee/refmash.html*

As part of the infrastructure 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

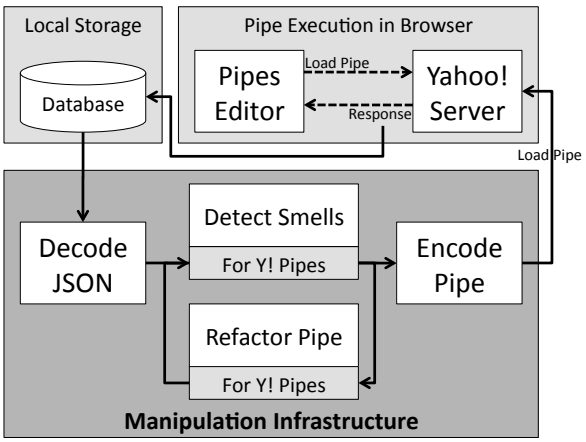

Figure 6: Study Infrastructure

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.

7.3 Refactoring Effectiveness Results

For each smell, Table 2 presents the number of smell instances (smelliness) per pipe in the *Smells Per Pipe* row, and each subsequent row shows the change in smelliness after applying each individual refactoring. For example, each pipe affected by the *Duplicate Modules* smell contains an average of 5.10 smelly modules. After applying the *Duplicate Paths* refactoring, each affected pipe has 1.43 smelly modules, a reduction of 72%.

Seven of the refactorings applied individually are able to completely remove certain smells from the pipes: *Non Contributing Module* eliminates *Unnecessary Module*, *Push Down Module* eliminates *Unnecessary Abstraction*, *Pull Up Module* eliminates *Duplicate Strings*, *Extract Local Subpipe* eliminates *Isomorphic Paths*, *Remove Deprecated Modules* eliminates *Deprecated Module*, *Normalize Module Ordering* eliminates *Non-conforming Module Orderings*, and *Extract Global Subpipe* eliminates *Global Isomorphic Path*. *Remove Deprecated Sources* is almost as effective, eliminating over 99% of the *Invalid Sources* smell.

We note that some refactorings cause changes that open the door for other refactorings to be performed. For example, the *Remove Deprecated Sources* refactoring not only eliminates 99% of the *Invalid Source* smells, but it also increases the presence of the *Unnecessary Module* smell by 25% (removing deprecated sources can lead to a module with no fields, fitting Smell 2.3). This creates an opportunity for *Remove Non-Contributing 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 the *Noisy Module* smell, with a maximum individual reduction of 18%, but a collective reduction closer to 43%.

We explore the effect of applying a sequence of refactorings utilizing a greedy approach to take advantage of the compounding effect of multiple refactorings. The results, shown in the last row of Table 2, 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. The *Noisy Module* smell is not eliminated because the implementation of *Refactoring 1: Clean Up Module* only targets the generator and setter modules. The *Duplicate Modules* smell 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. The *Invalid Source* smell is not eliminated because *Refactoring 9: Remove Deprecated Sources* does not remove sources within user-setter modules.

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

7.4 Generalizability and Validity

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.

We take a step in this direction by performing a manual inspection and analysis of the pipes available in the newer DERI Pipes repository. 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 (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% 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% of the pipes, *Smell 8: Invalid Source* impacts 9% of the pipes, and Smell 2: Unnecessary Module impacts 6% of the pipes. Each of these smells can be eliminated using the refactorings described in Section 6.

There are two other threats to the validity of our results related to the proposed refactorings. First, the refactorings presented in Section 6 are guaranteed to generate pipes that produce the same set of unique items (reflecting Opdyke's definition [19]; proof sketches are available [23]). 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 is 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. Second, although the presence of smells was shown to negatively impact end-user programmers, we did not assess the proposed refactorings in the hands of end users to understand whether and how they are adopted in practice.

8. CONCLUSION

End users are developing and sharing mashups in increasing numbers. However, a popular kind of mashup created by end users, pipe-like mashups, have 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 identi-

Table 2: 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
Red.	Clean Up Module	-18.37%									
	Non-Contributing		-100.00%								
	Push Down Module	-11.33%		-100.00%	-10.21%						
Con.	Merge Modules					-17.23%			-		
	Duplicate Paths					-72.00%					
Abs.	Pull Up Module	-12.72%		-47.37%	-100.00%						
	Local Subpipe				-11.70%		-100.00%				-23.00%
Dep.	Deprecated Module							-100.00%			-7.14%
	Deprecated Source		+24.66%						-99.19%		
Pop.	Module Ordering									-100.00%	
	Global Subpipe										-100.00%
Greedy Approach		-42.65%	-100.00%	-100.00%	-100.00%	-89.70%	-100.00%	-100.00%	-99.19%	-100.00%	-100.00%

fied the most prevalent smells in a population of 8,051 pipes, and have shown that end users prefer pipes that lack these smells. 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 unique to the mashup domain (e.g., remove deprecated sources), and also some 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% and reduce the average smell instances per pipe by almost 90%. Given these promising results, the next step is to study these refactorings in the hands of end users to better understand how they can be leveraged most effectively.

Acknowledgments

This work was supported in part by 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. We would like to thank the EUSES consortium members and the ICSE reviewers for their feedback on this work.

9. REFERENCES

[1] Apatar. http://www.apatar.com/, August 2009.

[2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA*, 2005.

[3] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective.

[4] 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. In *ICSE*, 2003.

[5] DERI Pipes. http://pipes.deri.org/, August 2009.

[6] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, 2009.

[7] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] L. Grammel, C. Treude, and M.-A. Storey. Mashup environments in software engineering. In *Web2SE '10*, 2010.

[9] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2:228–269, July 1993.

[10] IBM Mashup Center. http://www.ibm.com/software/info/mashup-center/, August 2009.

[11] M. C. Jones and E. F. Churchill. Conversations in developer communities: a preliminary analysis of the yahoo! pipes community. In *C&T 09*.

[12] JSON. http://www.json.org/, August 2009.

[13] H. Kegel and F. Steimann. Systematically refactoring inheritance to delegation in java. In *ICSE*, 2008.

[14] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing java classes. In *ICSE*, 2007.

[15] 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. In *SIGSOFT FSE*, 2008.

[16] J. Liu, D. S. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*, 2006.

[17] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. on Software Engineering*, 30(2), 2004.

[18] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Soft. Maintenance and Evolution*, 17(4).

[19] W. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[20] Yahoo! Pipes. http://pipes.yahoo.com/, July 2009.

[21] Plagger. http://plagger.org/trac, August 2009.

[22] C. Scaffidi, B. A. Myers, and M. Shaw. Topes: reusable abstractions for validating data. In *ICSE*, 2008.

[23] K. T. Stolee. Analysis and Transformation of Pipe-like Web Mashups for End User Programmers. Master's Thesis, University of Nebraska–Lincoln, June 2010.

[24] G. Sunyé, D. Pollet, Y. Le Traon, and J. Jézéquel. Refactoring UML models. *LNCS*.

[25] G. Taentzer, D. Müller, and T. Mens. Specifying domain-specific refactorings for andromda based on graph transformation. In *AGTIVE*, 2007.

[26] J. Wong and J. Hong. What do we "mashup" when we make mashups? In *WEUSE*, 2008.