6-2010

Analysis and Transformation of Pipe-like Web Mashups for End User Programmers

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

ANALYSIS AND TRANSFORMATION OF PIPE-LIKE WEB MASHUPS FOR END USER PROGRAMMERS

by

Kathryn T. Stolee

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Sebastian Elbaum

Lincoln, Nebraska

June, 2010

ANALYSIS AND TRANSFORMATION OF PIPE-LIKE WEB MASHUPS FOR END USER PROGRAMMERS

Kathryn T. Stolee, M. S.

University of Nebraska, 2010

Adviser: Sebastian Elbaum

Mashups are becoming increasingly popular as end users are able to easily access, manipulate, and compose data from several web sources. To support end users, communities are forming around mashup development environments that facilitate sharing code and knowledge. We have observed, however, that end user mashups tend to suffer from several deficiencies, such as inoperable components or references to invalid data sources, and that those deficiencies are often propagated through the rampant reuse in these end user communities.

In this work, we identify and specify ten code smells indicative of deficiencies we observed in a sample of 8,051 pipe-like web mashups developed by thousands of end users in the popular Yahoo! Pipes environment. We show through an empirical study that end users generally prefer pipes that lack those smells, and then present eleven specialized refactorings that we designed to target and remove the smells. Our refactorings reduce the complexity of pipes, increase their abstraction, update broken sources of data and dated components, and standardize pipes to fit the community development patterns. Our assessment on the sample of mashups shows that smells are present in 81% of the pipes, and that the proposed refactorings can reduce that number to 16%, illustrating the potential of refactoring to support thousands of end users developing pipe-like mashups.

ACKNOWLEDGMENTS

I would like to acknowledge and give sincerest thanks to all those who have played a role in the completion of this thesis. Support for an endeavor such as this comes from many places, each uniquely important and worthy of mention, though not all can be listed on just one page. I am so blessed to have wonderful family, friends, and colleagues, and I thank you all sincerely.

It goes without saying that none of this work would have been possible without the guidance of my advisor, Sebastian Elbaum. I cannot thank him enough for his support and dedication. I would also like to thank all the ESQuaReD faculty, most specifically those on my committee, Gregg Rothermel and Anita Sarma, for their thoughtful feedback and suggestions.

My deepest thanks goes to my family, and especially my husband Derrick, for his unfailing love and his ability to keep me grounded and focused. I would also like to thank my best girlfriends, Aimee and Stephanie, for the countless girls' nights and adventures that help us all keep our sanity! The students in the ESQuaReD lab deserve special acknowledgement, especially Elena, Brady, Charlie, Isis, Zhihong, and Tingting, who have kept me caffeinated and smiling.

I would also like to extend gratitude to my collaborators in the EUSES Consortium, who have helped to motivate and guide this work. Additionally, I would like to thank all the rest of the faculty and staff in CSE for all they do in supporting students during our graduate careers.

This work was supported in part by the EUSES Consortium through NSF-ITR 0324861 and 0325273, NSF-0915526, and CFDA#84.200A: Graduate Assistance in Areas of National Need (GAANN).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A *mashup* is an application that manipulates, composes, and reuses existing data sources or functionality to create a new piece of data or service [11]. A *web mashup* is a mashup application in which the sources of data or services reside on the web. For example, a web mashup could grab data from some RSS feeds (e.g., house sales, vote records, bike trails), join those data sets, filter them according to a specified criteria, and plot them on a map published at a website [36].

Mashups have become extremely popular as development environments make it possible for end users to quickly get, process, and glue data with powerful APIs. For example, Yahoo Pipes [29], one of the most popular mashup development environments and the particular target of our techniques and empirical studies, provides users with a drag and drop environment to create *pipes* by selecting and configuring predefined modules and connecting them with wires through which data flows. We refer to these as *pipe-like mashups*. Over 90,000 end users have created pipes since 2007 and over 5 million pipes are executed in the Yahoo's servers daily [15].

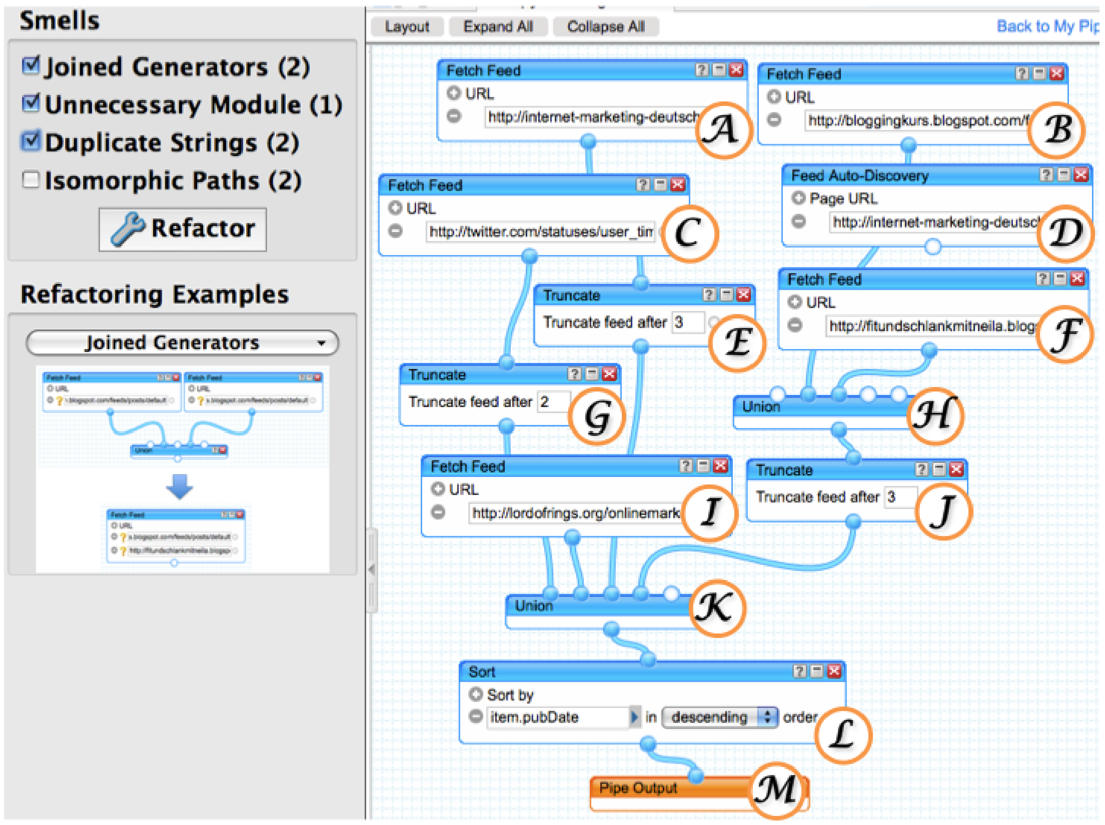
In addition to popular use by individuals, communities are forming around mashup environments that facilitate sharing of resources. Users are sharing their programs with one another via public repositories, sharing knowledge via message boards, and creating tutorials to help others learn. These trends of programming support and community growth for end user programmers can be seen across many mashup environments [1, 6, 14, 29, 37].

In spite of mashups' increasing power and popularity, in our investigation of pipe-like mashups we have observed that mashups 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. Clearly, these characteristics are undesirable and they become especially problematic when we consider that reuse is rampant among end user programmers. Over 73,000 pipes have been committed to Yahoo's public repository, and from the sample of 8,051 pipes we studied, 66% had been reused an average of 17 times. While reuse should be encouraged so users can take advantage of the knowledge and resources of the community, deficiencies are often propagated during the reuse process.

Undesirable program characteristics, such as the deficiencies described for mashups, are often referred to as code *smells*[10]. Smells are indications that something may be wrong with a piece of code. Professional programmers often perform semantic preserving transformations in their programs to remove smells and improve code quality, a process called *refactoring* [10]. While refactoring has been well-studied in the development environments utilized by professional programmers [28], these rich program transformation techniques have not yet penetrated the end user domain. Through the application of smell detection and refactoring techniques to web mashups, we can begin to address the deficiencies in web mashups and provide better support to end user programmers, allowing them to more effectively contribute to and utilize resources from their communities.

1.1 Motivating Example

We now present a running example to illustrate what is a pipe-like mashup, what are the potential smells in such mashups, and how refactorings can help to remove those smells.



Smells

- [✓ Joined Generators (2)]
- [✓ Unnecessary Module (1)]
- [✓ Duplicate Strings (2)]
- Isomorphic Paths (2)

Figure 1.1: Original Pipe

Figures 1.1 to 1.3 show screenshots of the Pipes Editor, Yahoo!'s development environment operating in a browser, alongside the prototype interface for our smell detection and refactoring toolset (grayed area). The Editor contains one of the pipes we collected from the Yahoo! Pipes repository. This pipe is meant to aggregate and sort news articles from several websites, and was selected for illustration due to the variety of smells it exhibits in a relative small number of modules. Throughout these figures we will see different revisions of this pipe as the refactorings are applied to address the smells detected (the letter labels

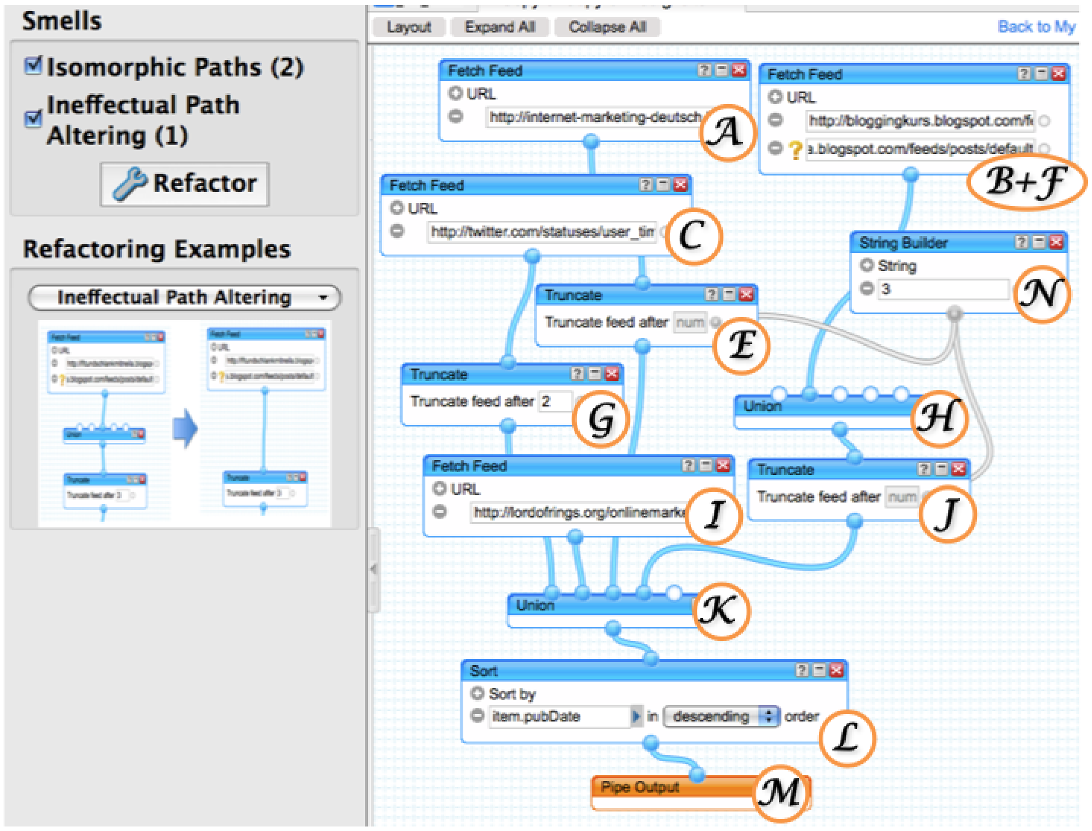


Figure 1.2: Partially Refactored Pipe

serve to reference the affected modules). At each stage, the prototype indicates the smells that are present in the pipe, shows examples of how the smells can be refactored, and offers to perform the refactorings for the user.

The structure of the pipe is best understood as a data-flow representation from top (inputs) to bottom (output). Each module is defined by the Pipe's environment, and is connected to other modules via wires. Most modules contain fields, and fields can contain hard-coded values or receive values via wire. In Figure 1.1, six generator modules retrieve data from external sources: five *Fetch Feed* modules, A, B, C, F, and I, and one *Feed Auto-Discovery* module, D, each containing one field. These generator modules provide the data for the rest of the pipe modules to process. Two generator modules, A and C, are wired directly into *Truncate* modules E and G, respectively. *Truncate* modules only retain the first n items to pass to the next module, where n is set by the field value. Two other generator modules, B and F, are aggregated through a path-altering *Union* module, H, before feeding to *Truncate* module J. Modules E, G, J, and I are aggregated with a *Union* module, K, that feeds to a *Sort* operator module, L, and finally to the pipe's *Output* module, M.

Although this is a functional pipe, it has several smells that can be removed by refactoring, while preserving the underlying semantics of the pipe. First, one of the modules, D, is completely disconnected from the pipe and serves no purpose, thus it should be removed. Additionally, the data produced by the two generator modules, B and F, are immediately aggregated prior to any manipulation. Since generator modules can accommodate multiple fields, this redundancy can be removed by joining them. Additionally, in two of the *Truncate* modules, E and J, the string "3" specifies the number of items to allow; 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 in future changes. Given that 36% of the pipes in our sample were modified after being published and 66% had been cloned at least once, making future changes easier is likely a concern for users.

The deficiencies and redundancies discussed for Figure 1.1 were addressed to generate the pipe in Figure 1.2. Generator modules B and F were merged to yield module $B + F$, the disconnected module D was removed, and a new module, N, was added to abstract the value "3" from modules E and J, which now receive their field values via wire. Despite these changes, the pipe in Figure 1.2 also suffers from some deficiencies. Since merging the generator modules, the *Union* module H has only one incoming wire, becoming ineffectual. Additionally, two of the paths leading to module K, A to E and C to G, are nearly identical except for their field values. These isomorphic paths can be abstracted into a separate pipe

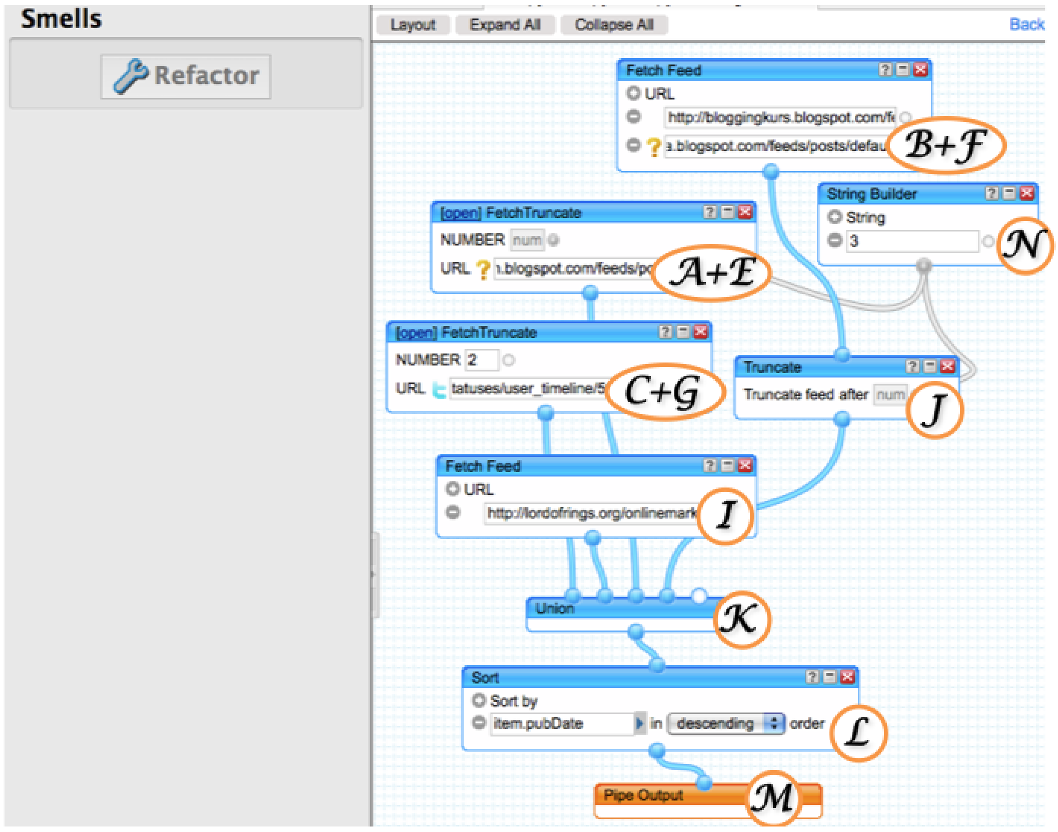


Figure 1.3: Completely Refactored Pipe

that can be included as *Subpipe* module, increasing the modularity and maintainability of the pipe.

Figure 1.3 shows the fully-refactored pipe from this example. Unnecessary module H was removed and a separate pipe was created to replace the isomorphic paths. Overall, 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.

The example just presented is not the only pipe that suffers from so many deficiencies. Of the thousands of pipe-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 we examined had at least one type of deficiency that made them more susceptible to failure, harder to maintain, or more difficult to reuse properly by other end users.

1.2 Research Contributions

To address these problems that limit the dependability of end user mashups and the productivity of end user mashup programmers, we adapt and extend a well-known software engineering technique, refactoring, to the mashup domain. Although the body of work on refactoring is rich, this is the first effort targeting the rapidly growing pipe-like mashup domain. Our contributions are:

- Identification of ten of the most prevalent smells observed in pipe-like mashups, their specification utilizing a general graph representation, and evidence that end users prefer non-smelly pipes over smelly pipes in nearly two-thirds of the exercises performed by 22 users in an empirical study.
- Design of eleven semantic preserving transformations to refactor a smelly pipe. We show how domain specific attributes such as the data-flow representation, the constrained set of parameterizable modules, and the reliance on the community, provide unique refactoring opportunities.
- Implementation of the analysis to detect the smells and the transformations to refactor pipes, tailored to the Yahoo! Pipes programming language.
- Study of the effectiveness of the refactorings on a population of 8,051 pipes developed by thousands of end users and shared through a public repository. The results indicate that the refactorings can eliminate the smells in 80% of the affected pipes.

1.3 Thesis Overview

After introducing end user programming, web mashups, and motivating the need for refactorings that can remove smells from mashup programs, the rest of the thesis is organized as follows. Chapter 2 presents related work pertaining to end user software engineering, mashups, refactoring, and graph transformations. Chapter 3 defines a family of code smells that are commonly found in web mashups as well as empirical evidence that users prefer non-smelly mashups to smelly mashups. We also present the results of a smell detection study that analyzes over 8,000 mashups created in the Yahoo! Pipes environment and shows that the smells we defined are indeed common. Chapter 4 defines refactorings that target the smells defined in Chapter 3, and shows the effectiveness of the refactorings by measuring the reduction in smells for the artifacts used in the smell detection study. Chapter 5 discusses the results of the studies and outlines several avenues for future work.

Chapter 2

Related Work

Although detecting smells in end user programs with removal through automated refactoring is novel, there are several avenues of related work that warrant discussion in this section. We briefly describe recent work in end user software engineering, the landscape of mashup environments, the emergence of end user communities surrounding mashup development tools, current work in traditional refactoring techniques, and the application of graph transformations to refactoring.

2.1 End User Software Engineering

In 2005 there were an estimated 55 million end user programmers in the United States, and that number has been projected to increase to 90 million in 2012 [32]. In general, end users create programs and engage in programming activities to support their hobbies and work, and the programs that are being created are meaningful to them, important to the businesses for which they work, and impactful to society. However, the programs created by end users suffer from the same deficiencies as those created by professional programmers (e.g., fault-proneness, security holes), making the quality of the programs created by end users a primary concern [12].

What differentiates end user programmers from professional programmers is that to end users, software is a means to an end, not the end itself. These end users utilize programming environments and languages such as spreadsheets, databases, web macros, mashups, and many domain-specific scripting languages. Additionally, many end users are also using languages and environments that were traditionally targeted at professional developers. For example, Amazon's Mechanical Turk command line interface requires knowledge of XML, web services, and shell scripting, yet it is targeted at "business analysts who do not want to write software" [25].

Unlike professional programmers, end users do not have much support for all stages of the software lifecycle. Researchers and practitioners have started applying software engineering techniques to provide support for end user tasks, yet these tools are far from pervasive in end user development environments. For example, version control has been introduced to help users during development [14], debugging has been introduced to allow users to ask questions about output during development [21] or preview program output during testing [29], assertions have been used to increase the dependability of web macros during runtime [22], and strides have been made toward providing better program maintenance by using program characteristics to predict how likely a program is to be reused [31]. However, considering the increasing number of end user programmers, the diversity of programs being created, and the deficiencies in their programs, providing better education and support for these end users is critical.

2.2 Mashups and Environments

Web mashups are a particular type of program that has gained popularity among end user programmers. Most broadly, a mashup is a program that takes several input sources, performs some operation, and creates a singular output. The target of the empirical studies in this work is Yahoo! Pipes [29], a mashup tool that will "retrieve data from one or several data sources, process the data, and publish the results as feeds or in widgets," otherwise referred to as an *information mashup* [11].

Many mashup development environments are available, some oriented toward more proficient developers that require users to know scripting languages (e.g., Plagger requires perl programming [30]), and many others are oriented toward environments and languages that allow users to work at higher levels of abstraction. These more abstract 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 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 [6], Feed Rinse [9], IBM Mashup Center [14], Kivati [20], Plagger [30], Yahoo! Pipes [29], and xFruits [37]). Some of these environments work exclusively with web data [6, 9, 29, 30, 37], while others allow the mashing of online and offline data sources [1, 14, 20]. Despite the variety and availability of different mashup environments that target a wide range of users and provide various types of abstraction and development support [36], over 90,000 users created mashups with Yahoo! Pipes between 2007 and 2009, making it one of the most popular environments available. [15].

Another interesting trend is the emergence of communities around these environments to provide user support, either as a forum, wiki, or better yet as a repository of mashups to be shared with other users [1, 6, 14, 29, 37]. The amount of support provided in each environment is in a large part dependent on the activity of the community, and researchers are only beginning to study these communities to discover how to better support end users developing mashups [15]. 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. From the examples of software engineering support for end user programming environments mentioned in Section 2.1, Yahoo! Pipes provides some level of debugging support and IBM Mashup Center provides some version control, but these solutions do not address all the correctness and quality concerns associated with end user programs, and this type of support is certainly not widespread in mashup environments [11].

2.3 Refactoring

Although no refactoring support has yet been introduced into mashup tools, the body of work on refactoring is extensive [28]. Refactorings are semantics-preserving transformations on source code, typically for the purpose of maintainability and simplicity. The decision to apply a refactoring is usually motivated by the presence of a smell, which is some characteristic in the code that could inhibit maintainability, introduce errors, or is unnecessarily complex. Fowler introduced refactoring to improve code design in object-oriented programs, and introduces a family of refactorings and smells in his seminal Refactoring book [10]. Many Integrated Development Environments (IDEs), such as Eclipse [8], include rudimentary refactoring support such as renaming methods or variables, extracting classes, and introducing parameters. However, other refactorings that, for instance, introduce design patterns or update calls to deprecated libraries, require more specialized support.

To provide more sophisticated refactoring support, researchers have applied refactoring techniques toward a number of targets. Toward improving code design and increasing maintainability, refactoring has been used to make java libraries more expressive and type safe by introducing generic type parameters [19], to migrate references to deprecated library classes using specifications that map legacy classes to replacement classes [2], to support parallelization of sequential code using program analyses that will introduce calls to libraries that support concurrency [7], and to making programs reentrant by replacing global state with thread-local state for the purpose of deploying on parallel machines [35]. To automate code updates, refactorings have been captured as they occur during the evolution of libraries, and then replayed to refactor and update client applications [13]. To generalize bloated code that results from unnecessary inheritance, delegation can be introduced to decouple inherited classes by applying the Delegation Design Pattern [18]. While the refactoring techniques just described are targeted at the library or source-code level, refactoring can also be used at a higher level of abstraction. For example, a process of feature oriented refactoring has been developed to decompose a legacy system into features by manipulating the program structure in an effort to support feature-based changes during program evolution [24].

The evaluations of these recent refactoring techniques have focused on languages utilized by professional software developers, and developer support has been integrated into IDEs such as Eclipse for many of the recent refactoring works [2, 7, 13, 18, 19, 35]. In these studies, a typical course of evaluation is to implement the refactorings in a tool and evaluate the tool on a set of programs, measuring time to completion [2, 19, 35], changes in program size [2, 18, 35], or the tool's accuracy when compared to the same refactorings performed manually [7, 19]. Part of our work follows a similar approach, taking advantage of a public mashup repository to perform a study of larger scale to determine the prevalence of the smells (Section 3.4) and the effectiveness of the refactorings in addressing those smells (Section 4.4).

While refactorings have been thoroughly studied, little effort has gone toward the precise definition and analysis of code smells. To some extent, refactoring preconditions indirectly create smell definitions, or at least define necessary conditions for a refactoring to be applied, yet providing such specifications is not widespread [18, 35]. In our work, we precisely define a family of code smells and demonstrate how the application of refactorings creates opportunities for other refactorings, and how multiple refactorings can work together to more completely target certain smells. In addition to an empirical analysis on the effectiveness of refactorings, we also explore the impact of the smells by conducting a user study that assesses whether smells matter to users (see Section 3.3).

2.4 Graph Transformations

Despite the rich literature on refactoring, program transformations are often expressed in natural language or by example, and this can lead to ambiguities and incomplete definitions that can under or over estimate their applicability [18]. Graph transformations have been shown to be suitable for expressing refactoring transformations, and can be used to prove the preservation of certain program properties [26] and to detect dependencies among different refactorings [27].

Although refactoring and model-driven software development are separate areas of research, recent work has explored the connections between the two areas, as one of the goals of both research areas is to manage software complexity [4]. 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 [33] or facilitate the transformation of different types of UML [34] or EMF [23] diagrams. Although not exclusively [26], it is among such model refactorings that we often see the use of graph transformations as a mechanism to make the preconditions, postconditions, and transformation steps explicit to work in complex software systems [3]. Influenced by such use of graphs to perform model transformations, we have adopted a graph-based notation to make explicit the smell detection and refactoring transformations described in this paper.

Chapter 3

Smell Detection

A *code smell* is a term used to refer to deficiencies in code. These program characteristics indicate something might be wrong with the code, and can be used to determine when code should be refactored. In Fowler's refactoring book, he provides motivation and guidelines for refactorings and smells, but specifications are not given for when refactoring should be performed. He argues that human intuition is the best indication of when refactoring should occur [10]. However, as demonstrated in recent work that automated the *Replace Inheritance with Delegation* refactoring, program transformations that are expressed in natural language or by example can lead to ambiguities and incomplete definitions that may cause refactorings to be confusing or less applicable than they appear [18]. In this section, we present precise definitions for code smells that exist in web mashups and can be used as preconditions for refactoring transformations.

We begin by presenting some general definitions for notation that represents a pipe-like mashup. These definitions will be referred to throughout the rest of this work. Then, we use those definitions to define 10 code smells we discovered by studying common deficiencies in mashups found in the Yahoo! Pipes repository. Next, we present the results of an empirical user study aimed to determine the impact of smells on the understandability

$$\boxed{\begin{array}{rl} in\_wire(m, w) & w.dest = m \land w.fld = \varnothing \\ out\_wire(m, w) & w.src = m \end{array}}$$

Figure 3.1: Shorthand for Common Predicates Used in Mashup Definitions

and maintainability of pipes. Finally, we describe our infrastructure for smell detection and present the results of another empirical study showing the commonality of the smells in a sample of 8,051 pipes programs from the Yahoo! Pipes repository.

3.1 Mashup Notation Definitions

A pipe-like mashup can be represented as a directed acyclic graph where the modules are nodes and the wires are the edges that transmit items between the modules in a pipe. Figure 3.1 presents some shorthand notations that are used in these definitions.

Definition 1. *A module is a tuple* $(\mathcal{F}, name, type)$*, containing a list of fields* $\mathcal{F}$ *indexed from* 1 *to* $\mid \mathcal{F} \mid$ *where* $\mathcal{F}[1]$ *is the first item in the list, a* $name$ *assigned by the Pipes programming environment (e.g.,* $fetch$ *or* $truncate$ *module names from Figure 1.1), 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 one module named* $output$.

Definition 5. *A pipe path is a sequence of* n *connected modules* $m_i \in \mathcal{M} \mid \forall m_i, 0 < i < n-1$, $\exists w \in \mathcal{W} \mid out\_wire(m_i, w)\ \wedge in\_wire(m_{i+1}, w)$. *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* $p(last)$.

A module's $m.type$ can be one of the following: 1) *generator* if it retrieves 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* if it either joins multiple paths, as in a union, or diverts one path into multiple paths, as in a split; and 4) *operator* if it obtains, manipulates, and produces a list of items. More formally, for $m \in \mathcal{M}$:

$$m.type = \begin{cases} gen & \texttt{if} \exists f \in m.\mathcal{F}, \exists es \in ExternalSources \mid \\ & \quad f \text{ refers to } es \\ setter & \texttt{if} \nexists w \in \mathcal{W} \mid in\_wire(m, w) \\ pathAlt & \texttt{if} \exists w_i, w_j \in \mathcal{W} \mid \\ & (in\_wire(m, w_i) \wedge in\_wire(m, w_j)) \vee \\ & (out\_wire(m, w_i) \wedge out\_wire(m, w_j)) \\ op & \texttt{if} \exists_1 w_i \in \mathcal{W} \mid in\_wire(m, w_i) \wedge \\ & \exists_1 w_j \in \mathcal{W} \mid out\_wire(m, w_j) \end{cases}$$

We further subtype a *setter* module as a *string-setter* if m sets a string ($m.type = setter.string$) or as a *user-setter* if the user may be queried to set a parameter when the pipe is executed ($m.type = setter.user$). An operator module o can be further classified across two orthogonal dimensions. First, o is said to be *read-only* if it does not modify the content of items in the input list ($o.type = op.ro$); o is said to be *read-write* if it can modify the content of list items, such as appending a string to the title of each list item ($o.type = op.rw$). 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 ($o.type = op.orderIndep$). This kind of module will, for example, sort, remove, and rename list items. On the other hand, o is *order-dependent* if the outcome depends on the order of the items passed into it ($o.type = op.orderDep$). An example is a *truncate* module that only outputs the first n items in a list.

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 item order (further details are provided in Appendix C). As a pipe's intention is to aggregate and manipulate data from multiple sources, duplicate entries do not provide new information to the user and the order of items that reach the $output$ module can be easily manipulated using a $sort$ module. In terms of semantics, order preservation is useful, but not necessary.

3.2 Smell Definitions

In this section, we define a family of code smells that can help an end user programming with Yahoo! Pipes identify areas where refactoring may be useful. Many of the smells were inspired by smells found in the code of professional programmers [10], and the smells target different aspects of a pipe-like mashup, including modules and fields that do not contribute to the pipe or are unnecessarily duplicated. Other smells were inspired by deficiencies we witnessed when exploring the population of Yahoo! Pipes and the availability of many ex-

$op(m)$	$m.type = op$
$op\_indep(m)$	$m.type = op.orderIndep$
$setter\_str(m)$	$m.type = setter.string$
$path\_alt(m)$	$m.type = pathAlt$
$gen(m)$	$m.type = gen$
$union(m)$	$m.name = union$
$in\_wire(m, w)$	$w.dest = m \wedge w.fld = \varnothing$
$field\_wire(m, w)$	$w.dest = m \wedge w.fld \neq \varnothing$
$out\_wire(m, w)$	$w.src = m$
$all\_field\_wires(m)$	$\forall w \in W \mid w.dest = m, field\_wire(m, w)$
$joined\_by(m_i, m_j, w)$	$out\_wire(m_i, w) \wedge in\_wire(m_j, w)$
$subsequent\_modules(m_i, m_j)$	$\exists$ path $p \mid m_i, m_j \in p \wedge m_i \prec m_j$
$between\_modules(m_k, m_i, m_j)$	$\exists$ path $p \mid m_k, m_i, m_j \in p \wedge m_i \prec m_k \prec m_j$
$connected\_to\_union(m_i, m_j)$	$\exists m_u \in \mathcal{M} \mid union(m_u) \wedge \exists w_i, w_j \in \mathcal{W}$ $\mid joined\_by(m_i, m_u, w_i) \wedge joined\_by(m_j, m_u, w_j)$
$same\_number\_fields(m_i, m_j)$	$\mid m_i.\mathcal{F} \mid = \mid m_j.\mathcal{F} \mid$
$all\_same\_field\_values(m_i, m_j)$	$same\_number\_fields(m_i, m_j)$ $\wedge$ for $k = 1 \cdots \mid m_i.\mathcal{F} \mid, m_i.\mathcal{F}[k] = m_j.\mathcal{F}[k]$
$same\_field\_value(f_i, f_j, bool)$	$f_i.wireable = f_j.wireable = bool$ $\wedge f_i.value = f_j.value \neq$ “”

Figure 3.2: Shorthand for Common Predicates Used in Smell Definitions

ample pipes generated by the community. These other smells identify modules that have been deprecated or patterns of modules that do not conform to standards set by the population. Figure 3.2 presents some shorthand definitions that are used throughout this section.

3.2.1 Laziness Smells

This category of smells was inspired in part by the *Lazy Class* smell, which aims to inline classes, components, or methods that do not do enough [10]. Similarly, 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: *a module that has unnecessary fields, making a pipe harder to read, less efficient to execute, and potentially adding errors that go unnoticed by the end user. Module $m \in \mathcal{M}$ is considered noisy if:*

Case 1.1. *Empty field:*

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

Case 1.2. *Duplicated field:*

$\exists f_i, f_j \in m.\mathcal{F} \mid same\_field\_value(f_i, f_j, true)$

Smell 2. Unnecessary Module: *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:*

$\exists n \in \mathcal{M} \mid n.name = output \wedge !subsequent\_modules(m, n)$

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:*

$!path\_alt(m) \wedge m.\mathcal{F} = \varnothing$

Case 2.4. *Unnecessary redirection:*

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

Case 2.5. *Swaying module:*

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

$(op(m) \wedge all\_field\_wires(m))$

For example, in the transformation from Figure 1.1 to Figure 1.2, module D was removed because it fit Case 2.1. From Figure 1.2 to Figure 1.3, module H was removed because it fit Case 2.2.

Smell 3. Unnecessary Abstraction: *a module that always performs the same operation on constant field values (fields that are not wired) may be unnecessary. Module $m \in \mathcal{M}$ is unnecessarily abstract if:*

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

3.2.2 Redundancy Smells

Duplicated code has been identified as the worst smell in programs written by professionals [10]. The redundancy smells identify pipes that have duplicated strings, modules, or paths of modules. Redundancies in pipes bloat the modules and the pipe structure, adding unnecessary complexity, and making pipe understanding and maintenance more difficult.

Smell 4. Duplicate Strings: *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:*

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

$same\_field\_values(f_i, f_j, true)$

For example, in Figure 1.1 the truncate modules E and J have a duplicate string "3."

Smell 5. Duplicate Modules: *operator modules appearing in certain patterns may be redundant and candidate 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 order-independent operators:*

$op\_indep(m_i) \wedge \exists w \in \mathcal{W} \mid joined\_by(m_i, m_j, w)$

Case 5.2. *Consecutive path-altering modules:*

$$path\_alt(m_i)\ \wedge \exists w \in \mathcal{W} \mid joined\_by(m_i, m_j, w)$$

Case 5.3. *Joined generators:*

$$gen(m_i) \wedge connected\_to\_union(m_i, m_j)$$

Case 5.4. *Identical subsequent operators:*

$$op\_indep(m_i) \wedge all\_same\_field\_values(m_i, m_j)\wedge$$
$$subsequent\_modules(m_i, m_j)\wedge$$
$$\forall m_k \in \mathcal{M} \mid between\_modules(m_k, m_i, m_j)\wedge$$
$$(op\_indep(m_k) \vee union(m_k))$$

Case 5.5. *Identical parallel operators:*

$$op\_indep(m_i) \wedge all\_same\_field\_values(m_i, m_j)\wedge$$
$$connected\_to\_union(m_i, m_j)\wedge$$
$$\exists m_k, m_l \in \mathcal{M}, \exists w_k, w_l \in \mathcal{W} \mid$$
$$joined\_by(m_k, m_i, w_k) \wedge joined\_by(m_l, m_j, w_l)\wedge$$
$$(gen(m_k) \vee union(m_k)) \wedge (gen(m_l) \vee union(m_l))$$

For example, Case 5.1 is illustrated in Figure 4.2, Case 5.2 in Figure 4.3, Case 5.3 in Figure 4.5, Case 5.4 in Figure 4.4, and Case 5.5 in Figure 4.6.

Smell 6. Isomorphic Paths: *non-overlapping paths with the same modules performing the same manipulations 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* $k = 1 \cdots \mid m_n.\mathcal{F} \mid$
    *if* $m_n.\mathcal{F}[k].wireable = false$ *then* $same\_field\_value(m_n.\mathcal{F}[k], m'_n.\mathcal{F}[k], false)$
```

An example is shown in Figure 1.2, where p consists of the path from A to E and p' consists of the path from C to G.

3.2.3 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: *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: *a 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.*

3.2.4 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 pipes repositories 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 discourage reuse of pipes across the community.

Smell 9. Non-conforming Module Orderings: *given a community prescribed order for read-only and order-independent operator modules appearing in a path of size 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. We obtain a pool of 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.

Smell 10. Global Isomorphic Paths: *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

Generating $PGPaths$ requires identification of the population sample from which the paths are derived and a threshold of how often a path must occur to be considered global.

3.3 End Users' Perspectives on Smelly Pipes

In the software engineering literature, code smells are known to be things to avoid, and when they occur, removed through refactoring [10]. While a professional developer may be able to detect code smells, it has yet to be explored whether end users have the same awareness, and specifically in the context of pipe-like web mashups. The goal of this study is to evaluate the impact of coding practices, specifically smells, on the preference and understandability of pipe-like web mashups from the perspective of users in the context of the Yahoo! Pipes environment. We aim to answer the following question: *Are pipes with bad program characteristics (i.e., smells) less understandable or desirable than pipes without such characteristics?*

3.3.1 Study Design

We designed two experiments that evaluate the impact of code smells from the perspective of the user. The first aims to determine if users prefer clean [1] or dirty [2] pipes, presenting the user with two versions of the same pipe side-by-side, one clean and one dirty, and asking the user to select the most preferable. The second aims to determine if smelly or clean pipes are harder to understand by presenting the user with a pipe and a set of potential outputs, asking them to select the most fitting output. Each experiment is split into a series of tasks, as shown by the experimental design in Figure 3.3. In each experimental task, we treat one pipe with a *smell*, X. In the tasks associated with the first experiment (1–8), the user is presented with both the treated (smelly) and the untreated (clean) pipe. In the tasks for the second experiment (9–10), the user is either presented with the treated or the untreated pipe. In both experiments we estimate the user aptitude by measuring education level (O_1) and

[1]The terms *clean*, *non-smelly*, and *refactored* are used interchangeably.

[2]The terms *dirty* and *smelly* pipe are used interchangeably.

qualification score (O_2), using a comprehensive pretest. The first experiment also measures user preference (O_3), the second measures correctness (O_4), and both measure time (O_5).

Task	Assignment	Pretest Measures	Object	Treatment	Posttest Measures
1	R	O_1, O_2	P_1	X_5	O_3, O_5
2	R	O_1, O_2	P_2	X_4	O_3, O_5
3	R	O_1, O_2	P_3	X_5	O_3, O_5
4	R	O_1, O_2	P_4	X_8	O_3, O_5
5	R	O_1, O_2	P_5	X_7	O_3, O_5
6	R	O_1, O_2	P_6	X_1	O_3, O_5
7	R	O_1, O_2	P_7	X_5, X_{10}	O_3, O_5
8	R	O_1, O_2	P_8	X_2, X_7	O_3, O_5
9	R	O_1, O_2	P_9	X_6	O_4, O_5
	R	O_1, O_2	P_9		O_4, O_5
10	R	O_1, O_2	P_{10}	X_2, X_3	O_4, O_5
	R	O_1, O_2	P_{10}		O_4, O_5

Figure 3.3: Experimental Design

Each smell defined in Section 3.2 is used as a treatment (X_n) in at least one task, where the subscripted number, n, represents a particular smell. All the pipes (P_m) used in the experiments were derived from pipes found in the public Yahoo! Pipes repository, and each task uses a different pipe (indicated by different values of m). A pipe was selected only if the pipe structure was dirty; the clean version was generated by manipulating the pipe to remove the smell. In this way, the smells were representative of real smells generated by users in the Yahoo! Pipes community, and were not artificially inserted. Some pipes were also modified for size so the study participants would not be overwhelmed when trying to understand each pipe's behavior. Size modifications included removal of modules, wires, and fields, but nothing was added.

Users self-selected their participation for each task in the experiment, giving a random assignment of participants to tasks (represented by the *R* in Figure 3.3). Our goal was to consider participants from a range of backgrounds and expertise to be representative of end users, and control for variability by segmenting the population based on user aptitude in the analysis. Prior to participation, all users were required to pass a pretest test to demonstrate a basic knowledge of Yahoo! Pipes. We designed the test with eight comprehensive questions detailed in Appendix A.1.3. The passing score was set to 50%, a seemingly low-bar, but the test was designed to be challenging even for advanced users of Yahoo! Pipes. A tutorial video was provided for participants who were not previously familiar with the Yahoo! Pipes environment. The results of this pretest were used to measure user aptitude (O_1 and O_2).

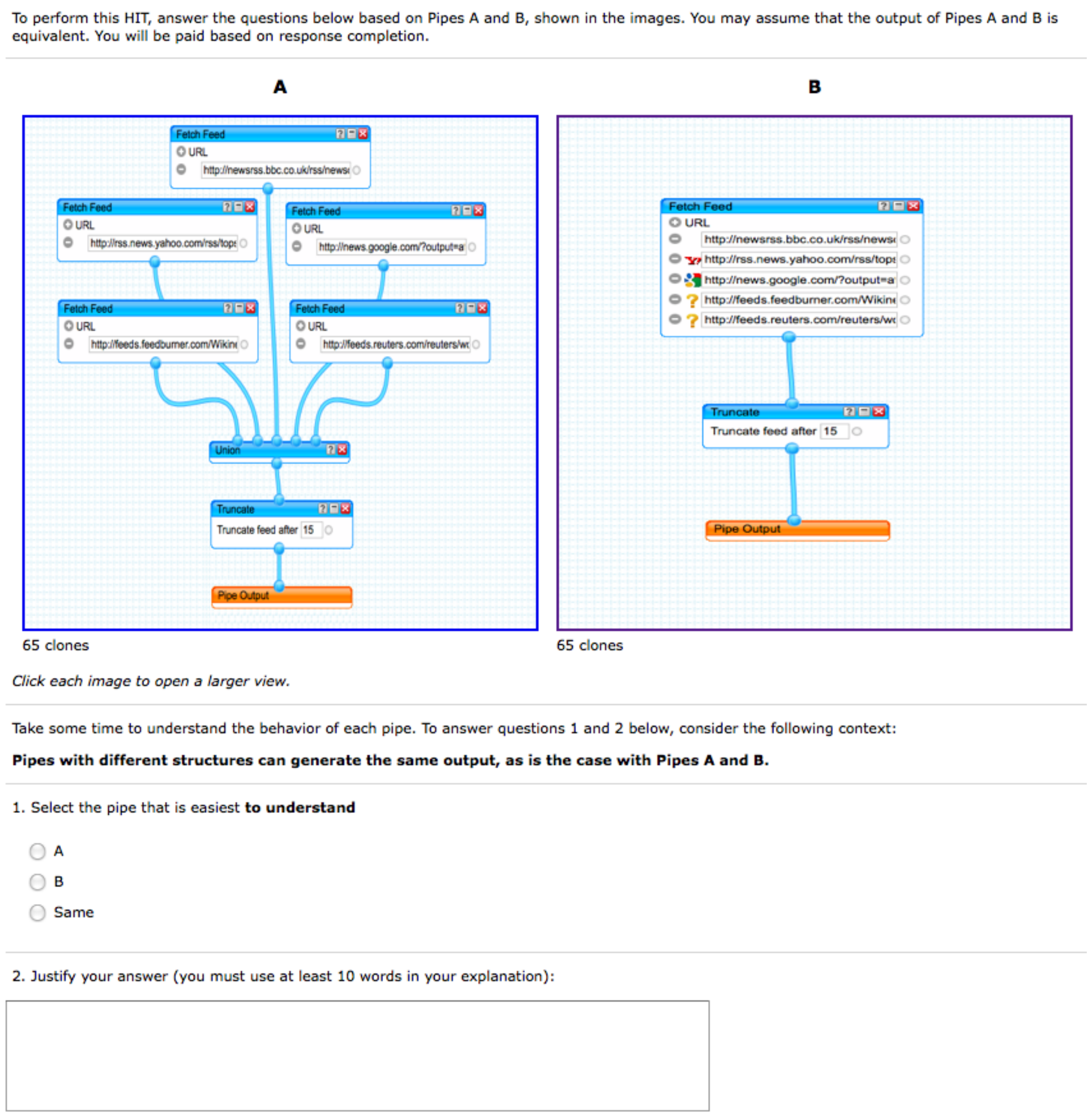
3.3.1.1 Experiment #1: Preference

Each task in this experiment was designed to assess a user's preference between two pipes given some context. Figure 3.4 gives an example of a task in this experiment, specifically representing task 1.

Pipe A represents P_1 with a dirty structure that is treated with smell X_5, defined by Smell 5.3:*Joined Generator*; Pipe B represents P_1 with a clean structure. The context reads, *Pipes with different structures can generate the same output, as is the case with Pipes A and B.* The user is asked to select the pipe that is easiest *to understand* and to justify their selection in short-answer form.

In Table 3.1, we identify the preference question, map each smell treatment to its named smell definition, indicate the size of each the clean and smelly versions of the pipe structure in terms of the number of modules, and give the context for each preference task. The size is given as a proxy for the complexity of the pipes we asked the participants to analyze.

We offer coverage of different pipe types by applying different smell treatments for each task; as a result, each task presents different challenges to the participant. Some of the pipes contain complex structures, such as those for tasks 2 and 8, while others contain complex modules, such as the *loop* modules in task 5, the *regex* modules in task 3, and the *subpipe* in task 7. Many of the smelly pipes contain multiple paths, as is the case


Fetch Feed
URL
http://newsrss.bbc.co.uk/rss/news
http://rss.news.yahoo.com/rss/tops
http://news.google.com/?output=a
http://feeds.feedburner.com/Wikin
http://feeds.reuters.com/reuters/w
Union
Truncate
Truncate feed after 15
Pipe Output

To perform this HIT, answer the questions below based on Pipes A and B, shown in the images. You may assume that the output of Pipes A and B is equivalent. You will be paid based on response completion.

A **B**

65 clones 65 clones

Click each image to open a larger view.

Take some time to understand the behavior of each pipe. To answer questions 1 and 2 below, consider the following context:

Pipes with different structures can generate the same output, as is the case with Pipes A and B.

1. Select the pipe that is easiest **to understand**

- A
- B
- Same

2. Justify your answer (you must use at least 10 words in your explanation):

Figure 3.4: Preference Task Example

with tasks 1, 2, and 4, while others contain modules with many parameters, for example in tasks 6 and 7.

Table 3.1: Preference Tasks General Description

Task	Question	Smell Treatment	Dirty Size	Clean Size	Context
1	to understand	Joined Generators	8	3	Pipes with different structures can generate the same output, as is the case with Pipes A and B.
2	to update in the future	Duplicate Strings	12	13	Truncate modules in Pipe A have hard-coded field values, but receive values via wire in Pipe B.
3	to understand	Consecutive Operators	7	6	Rules in Regex modules modify a specified field's content (e.g., item.title), replacing instances of a pattern ($\hat{(}$.+)) with some text (JENI Latest -).
4	to update in the future	Invalid Source	8	6	Websites can be deleted, causing 404 errors, like these 2 in Pipe A: http://www.gamemakergames.com and http://www.gmshowcase.dk/forums.
5	for others to understand	Deprecated Module	4	4	Components are sometimes deprecated and replaced with improved features. In Pipe B, Content Analysis and For Each: Replace were deprecated.
6	to update in the future	Noisy Module	3	3	Specifying the same website multiple times can lead to duplicate items in a pipe's output.
7	to understand	Global Isomorphic Paths	5	3	In Pipe A, the ”Fetch 6, Unique” subpipe module gathers the content of the six specified URLs and removes items that have duplicate titles or links.
8	for others to understand	Module Orderings	5	4	The majority of the most popular pipes in the Yahoo! Pipes repository place the Unique module before the Filter module.

3.3.1.2 Experiment #2: Output Analysis and Correctness

The output tasks are designed to determine if users can understand the behavior of a Pipe and to assess how understandable non-smelly pipes are compared to smelly pipes. The experiment contains two tasks. To meet the first goal, each task gives the user an image of a pipe and asks the user to select which multiple-choice answer best describes the output of the pipe. We achieve the second goal by showing the experimental group participants the treated pipe and showing the control group participants the clean pipe.

In an output analysis task, the user is given a pipe image, as shown in Figure 3.5, which maps to the control group in Task 9, showing P_9 with a clean structure. The context reads, *Subpipe ”fetchfilterpermitany” behaves as follows: it gathers the content of the website specified in the Feed field and filters items based on the title or description*, and the user is asked to select the answer to a multiple-choice question that most appropriately represents the pipe's output, followed by a short-answer justification.

In Table 3.2, for each pipe in each task, we present the size of each pipe in terms of the number of modules, the smell treatment mapped to its smell name, and the context for each task in each experiment.

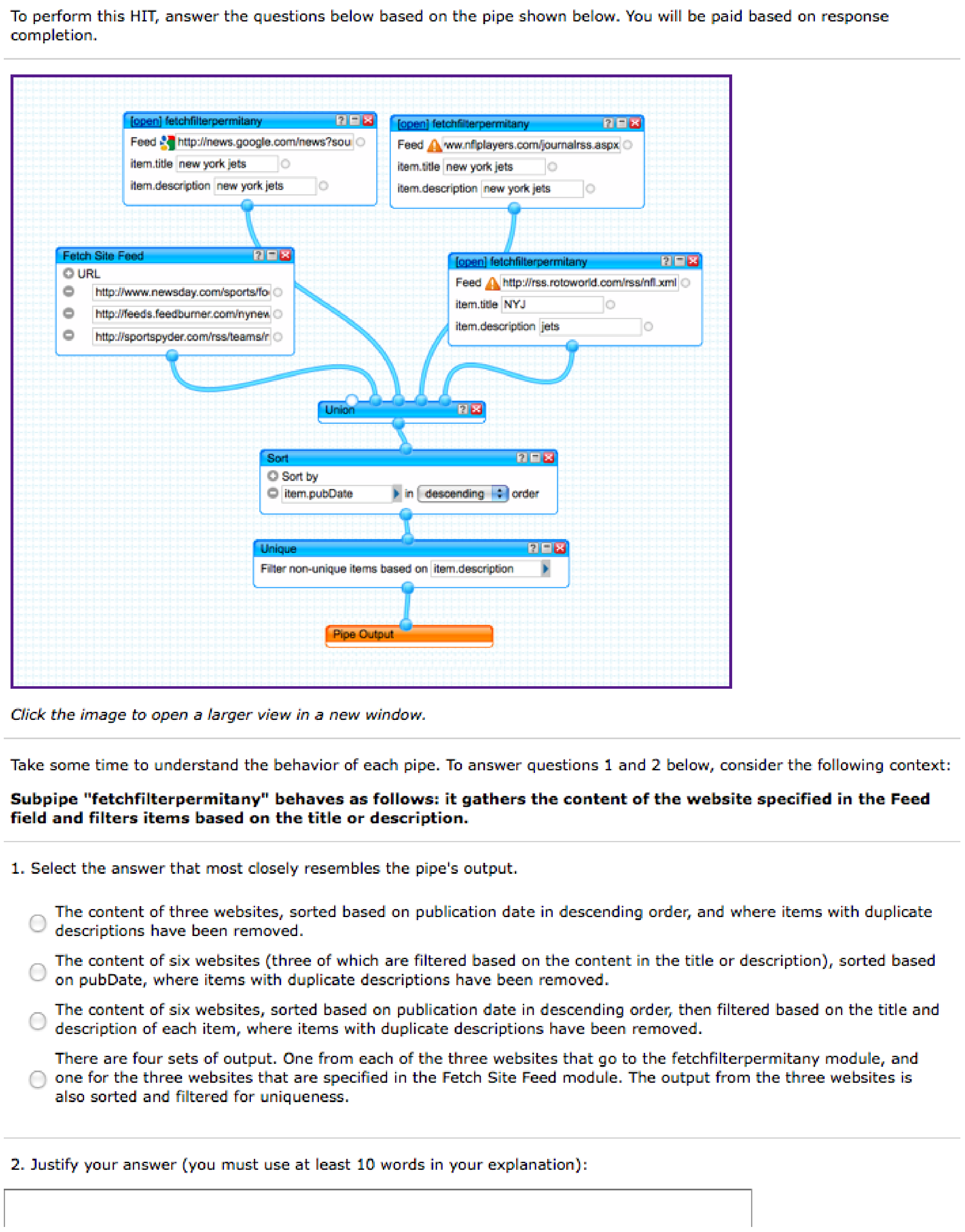
Table 3.2: Output Analysis Tasks General Description

Task	Group	Size	Smell Treatment	Context
9	Control	8	none	A filter module can be configured to permit or block items with certain characteristics. When multiple rules are provided, the filter module can consider any or all of the rules.
	Experimental	11	Isomorphic Paths	
10	Control	5	none	A Search For module is a user-input module that gets a string from the user when the Pipe is run, using that string to set the value of fields connected via wire.
	Experimental	11	Unnecessary Module & Unnecessary Abstraction	

Each of the experimental tasks has a different smell treatment that creates unique challenges for the user. The pipes for experimental groups in Tasks 9 and 10 contain complicated structures with multiple paths through the pipe. Both pipes for Task 10 contain complex modules, such as user-input modules, and the pipe in the control group for Task 9 contains *subpipe* modules.

3.3.1.3 Presentation of Results

The results are presented in summary across the participants, organized per experiment and then per task. We also use the Spearman's Rank Correlation Coefficient to explore relationships between the measures in the experiments, such as education (O_1) and selection of the non-smelly pipe (O_3) in the first experiment. This statistical measure is a non-parametric value that indicates the linear relationship between two variables, assigning a value between



To perform this HIT, answer the questions below based on the pipe shown below. You will be paid based on response completion.

[open] fetchfilterpermitany
Feed http://news.google.com/news?sou
item.title new york jets
item.description new york jets

[open] fetchfilterpermitany
Feed ww.nflplayers.com/journalrss.aspx
item.title new york jets
item.description new york jets

Fetch Site Feed
URL
http://www.newsday.com/sports/fo
http://feeds.feedburner.com/nynew
http://sportspyder.com/rss/teams/r

[open] fetchfilterpermitany
Feed http://rss.rotoworld.com/rss/nfl.xml
item.title NYJ
item.description jets

Union

Sort
Sort by
item.pubDate in descending order

Unique
Filter non-unique items based on item.description

Pipe Output

Click the image to open a larger view in a new window.

Take some time to understand the behavior of each pipe. To answer questions 1 and 2 below, consider the following context:

Subpipe "fetchfilterpermitany" behaves as follows: it gathers the content of the website specified in the Feed field and filters items based on the title or description.

1. Select the answer that most closely resembles the pipe's output.

- ○ The content of three websites, sorted based on publication date in descending order, and where items with duplicate descriptions have been removed.
- ○ The content of six websites (three of which are filtered based on the content in the title or description), sorted based on pubDate, where items with duplicate descriptions have been removed.
- ○ The content of six websites, sorted based on publication date in descending order, then filtered based on the title and description of each item, where items with duplicate descriptions have been removed.
- ○ There are four sets of output. One from each of the three websites that go to the fetchfilterpermitany module, and one for the three websites that are specified in the Fetch Site Feed module. The output from the three websites is also sorted and filtered for uniqueness.

2. Justify your answer (you must use at least 10 words in your explanation):

Figure 3.5: Output Analysis Hit Example, as it Appears to a Participant

-1 and $+1$, where values close to 0 indicate a low correlation. Our correlation analysis was performed using Apache's *org.apache.commons.math.stat* library [5].

3.3.2 Study Implementation and Operation

This study was implemented using Amazon's Mechanical Turk website [25], a service provided by Amazon 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. The worker is the one who completes the HIT and gets paid for their work, if satisfactory.

Each of the tasks described in Section 3.3.1 was implemented as a HIT, and users were paid $0.20 per HIT completed. To deliver the pretest, we created a custom qualification test that was required before a user could complete any HITs (see Appendix A.1). Once a user submitted a qualification test, it was graded as per our specification. A passing score allowed the user to complete the HITs we created. Participants were given a maximum of 60 minutes to complete each HIT and the study was available for two weeks, from April 28 - May 13, 2010. We show the workflow for a user participanting in our study in Figure 3.6. A user must first create an account in Mechanical Turk and then locate our HITs by searching. Figure 3.7 shows our tasks as they would appear to participants searching for tasks to complete. Next, they must take the qualification test. If they pass the test, the participant is able to perform any of the HITs in this study.

The initial participant recruitment relied on people finding our HITs on the Mechanical Turk website. After the first week, we began soliciting participation using internal mailing lists. While our targeted recruitment undoubtedly increased participation in the study, we

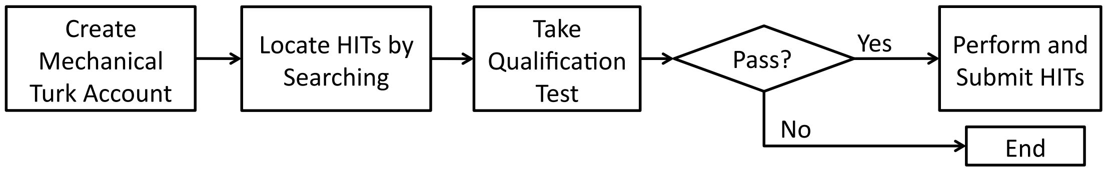
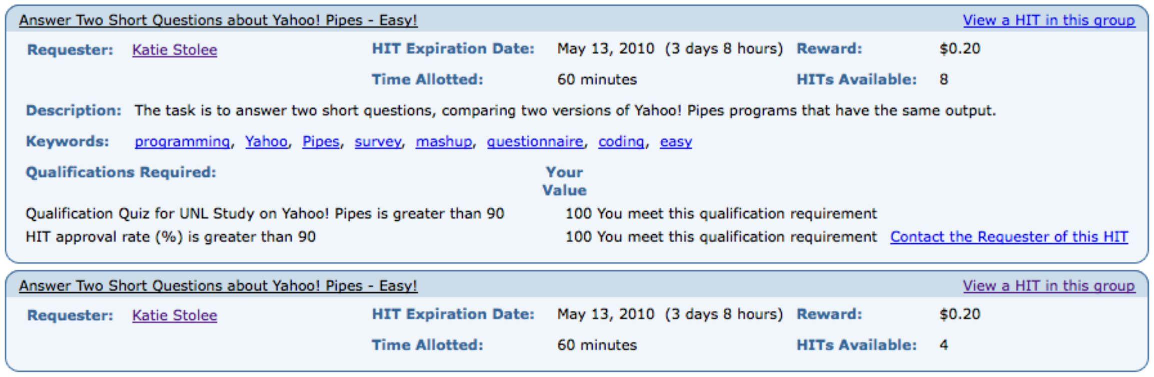


Figure 3.6: Participant Workflow in Mechanical Turk



Answer Two Short Questions about Yahoo! Pipes - Easy! View a HIT in this group

Requester:	Katie Stolee	HIT Expiration Date:	May 13, 2010 (3 days 8 hours)	Reward:	$0.20
		Time Allotted:	60 minutes	HITs Available:	8

Description: The task is to answer two short questions, comparing two versions of Yahoo! Pipes programs that have the same output.

Keywords: programming, Yahoo, Pipes, survey, mashup, questionnaire, coding, easy

Qualifications Required:	Your Value	
Qualification Quiz for UNL Study on Yahoo! Pipes is greater than 90	100 You meet this qualification requirement	
HIT approval rate (%) is greater than 90	100 You meet this qualification requirement	Contact the Requester of this HIT

Answer Two Short Questions about Yahoo! Pipes - Easy! View a HIT in this group

Requester:	Katie Stolee	HIT Expiration Date:	May 13, 2010 (3 days 8 hours)	Reward:	$0.20
		Time Allotted:	60 minutes	HITs Available:	4

Figure 3.7: Our HIT Entries in Mechanical Turk

are unable to distinguish between participants who found the HITs on their own and those whom we contacted, since Amazon anonymizes the identities of all participants.

3.3.3 Participant Profiles

While 50 users took the qualification quiz and 34 (68%) received a passing score, a total of 22 users completed at least one HIT in the study. We received 160 HIT responses, and each participant completed an average of 7.3 HITs. Although we did not control for gender in this study, we did collect that data. Among the 22 participants there were 12 females and 10 males. The users had varying levels of education in computer science, as is shown by their responses to the question, *What is your programming experience?* in Table 3.3. The *% of Population* column indicates what percentage of the total participants selected that answer for their programming experience, and the *% Male* and *% Female* columns indicate what percentage of those respondents were male or female, respectively. In terms of years of experience programming, the users declared similarly varied responses, as shown in Table 3.4.

Table 3.3: User Responses to the Question, *What is your programming experience?*

Response	**#**	**% of Population**	**% Female**	**% Male**
Limited or No Experience	5	24%	80%	20%
Self-taught only	3	14%	67%	33%
On-the-job training only	0	0%	0%	0%
One or more classes, in high school or college, in computer science or related field	6	24%	83%	17%
Undergraduate/Graduate degree, in progress or completed, in computer science or related field	8	38%	13%	87%
Total	**22**	**100%**	**55%**	**45%**

Table 3.4: User Responses to the Question, *How long have you been programming (e.g., using languages like Java, C/C++, JavaScript, Perl, Python, etc.)?*

Response	**#**	**% of Population**	**% Female**	**% Male**
no experience	5	24%	80%	20%
less than 1 year	4	19%	75%	25%
1-5 years	6	14%	67%	33%
5+ years	7	33%	14%	86%
Total	**22**	**100%**	**55%**	**45%**

For the purpose of analysis, we measure user aptitude in two ways, according to *education* (O_1) and according to *qualification score* (O_2). The first division is based on the idea that users with degrees in computer science will have more aptitude for the tasks than those without degrees. The second division is based on our intuition that formal education does not necessarily imply more attention to detail for these tasks, and we can estimate a participants' attention to detail using their qualification score. It seems appropriate to use both measures given that the Spearman rank correlation coefficient between education and scores on the qualification test indicates a low correlation ($r = 0.2164$).

In our measure of education (O_1), we identify two groups, *degreed users* and *end users*. We define the *degreed users* as those who have a degree in computer science or related field (last answer in Table 3.3), and the *end user* group as those who do not have a degree in computer science. We compare other characteristics between the two groups in Table 3.5.

Table 3.5: Comparison Between Groups Segmented on Education (O_1)

Characteristic	**End Users**	**Degreed Users**
Number of Participants	14	8
Total HITs Completed	91	69
HITs Completed per User	6.50	8.63
Percentage (female | male)	(79% | 21%)	(13% | 87%)
Average Qualification Score	79%	78%
Median Qualification Score	88%	94%

As shown in Table 3.5, among the 160 HITs completed by the users in this study, 91 were completed by the end users and 69 were completed by degreed users. The average scores on the qualification test among the groups of degreed users and end users are very similar, as was expected based on the low Spearman rank correlation coefficient ($r = 0.2164$) between education and qualification test scores. For both groups, the median scores on the qualification test were higher than the averages, showing that most participants scored above the average. In fact, only seven users out of 22 scored below 75%. The end users scored an average of 79% on the test, whereas the degreed users scored an average of 78%. This also shows a similarity in the collective understanding of Yahoo! Pipes between the two groups, making a comparison of the responses appropriate. The gender split between the end user and degreed user groups is quite disparate, with the end user group being composed of nearly 80% female, and the degreed user group being composed of 87% male participants.

For the purpose of measuring qualification test scores (O_2), we split the population into three groups of relatively equal size based on their scores. The *High Score* group received a perfect score on the qualification test, the *Middle Score* group scored in the upper quartile but less than perfect, and the *Low Score* group scored less than 75%.

Table 3.6: Comparison Among Groups Segmented on Qualification Score (O_2)

Characteristic	**High Score**	**Middle Score**	**Low Score**
Qualification Score	100%	75% - 88%	$<$ 75%
Number of Participants	9	6	7
Total HITs Completed	70	44	46
HITs Completed per User	7.78	7.33	6.57
Percentage (female | male)	(44% | 56%)	(67% | 33%)	(57% | 43%)

Table 3.6 presents the group characteristics of participants split based on qualification test scores. Unlike the segmentation strategy according to education, the ratio of female to male participants in these groups is similar to the ratio in the population as a whole. The Spearman rank correlation coefficient between gender and qualification scores also indicates very low correlation, with $r = -0.0601$.

3.3.4 Results

For the tasks in the first experiment, we measure user preference (O_3) between the non-smelly pipe, the smelly pipe, or do not express a preference (neutral). For the tasks in the second experiment, we measure correctness (O_4) of the answers and compare the output results of of the control group versus the experimental group. Overall, we see that participants preferred the non-smelly pipes in 61% of the first experiment tasks (an additional 18% of the responses were neutral), and were able to select the correct output for a pipe in

87% of the tasks in the second experiment (86% correct on the clean pipes and 88% correct on the smelly pipes).

3.3.4.1 Experiment #1: Preference

Table 3.7 presents the preferences (O_3) summarized across all tasks, organized by group according to the pretest measures. The *Segment* column indicates the measure used to split the population (i.e., based on education (O_1) or qualification score(O_2)), the *Group* column indicates the specific user group, and *Responses* indicates the total number of preference HIT responses received from a particular group. The *% Smelly* column indicates how many participants preferred the smelly pipe, *% Non-Smelly* indicates how many participants preferred the clean pipe, and *% Same* indicates the percentage of participants who were neutral.

Table 3.7: Overall Preference Selection

Segment	Group	Responses	% Smelly	% Non-Smelly	% Same
O_1	End Users	75	24%	63%	13%
	Degreed Users	55	18%	58%	24%
O_2	High Score	56	16%	63%	21%
	Middle Score	35	20%	63%	17%
	Low Score	39	31%	56%	24%
All Participants		130	22%	61%	18%

We further break down the user responses per task, shown in Table 3.8. In most cases, the overall preferred pipe among the participants was the non-smelly pipe, with few exceptions. We also explore the preferences per task (O_3) against the pretest measures of user aptitude (O_1 and O_2) and utilize these breakdowns in our analysis that follows.

The preference results for each task grouped by education level (O_1) are shown in Table 3.9. The *Smell Type* column indicates the category of the smell treatment applied in task, as defined in Section 3.2. The # column indicates the number of participants who

Table 3.8: Overall Preference Results per Task

Task	Responses	% Smelly	% Non-Smelly	% Same
1	17	12%	88%	0%
2	17	12%	88%	0%
3	17	18%	53%	29%
4	15	13%	27%	60%
5	15	47%	47%	7%
6	19	5%	63%	32%
7	15	67%	20%	13%
8	15	7%	93%	0%

Table 3.9: Preference Results for Based on Education (O_3 and O_1)

Task	Smell Type	End Users				Degreed Users			
		#	% Smelly	% Non-Smelly	% Same	#	% Smelly	% Non-Smelly	% Same
1	Redund.	10	10%	90%	0%	7	14%	86%	0 %
2	Redund.	10	10%	90%	0%	7	14%	86%	0 %
3	Redund.	10	20%	50%	30%	7	14%	57%	29 %
4	Environ.	8	25%	38%	38%	7	0%	14%	86 %
5	Environ.	9	56%	33%	11%	6	33%	67%	0 %
6	Lazy.	11	9%	73%	18%	8	0%	50%	50 %
7	Pop-based Redund.	9	56%	33%	11%	6	83%	0%	17%
8	Pop-based Lazy	8	13%	88%	0%	7	0%	100%	0%

submitted results for the particular task. Results for the preference tasks are also presented in Table 3.10, this time separated into groups based on qualification test scores.

For Task 1, a majority of the end users and the degreed users selected the non-smelly pipe as the most understandable. All of the end users who selected the non-smelly pipe indicated that they prefer pipes with fewer modules, mentioning that the non-smelly pipe is “cleaner,” “simpler,” and “looks less intimidating.” Similarly, the degreed users who

Table 3.10: Preference Results Based on Qualification Scores (O_3 and O_2)

Task	Smell Type	High Score				Middle Score				Low Score			
		#	% Smelly	% Non-Smelly	% Same	#	% Smelly	% Non-Smelly	% Same	#	% Smelly	% Non-Smelly	% Same
1	Redund.	6	0%	100%	0%	4	25%	75%	0 %	7	14%	86%	0%
2	Redund.	6	0%	100%	0%	5	0%	100%	0 %	6	33%	67%	0%
3	Redund.	8	13%	50%	38%	5	0%	100%	0 %	4	50%	0%	50%
4	Environ.	7	14%	14%	71%	4	25%	0%	75 %	4	0%	75%	25%
5	Environ.	7	43%	57%	0%	4	50%	50%	0 %	4	50%	25%	25%
6	Lazy.	8	0%	75%	25%	5	0%	40%	60 %	6	17%	67%	17%
7	Pop-based Redund.	7	57%	14%	29%	4	75%	25%	0%	6	75%	25%	0%
8	Pop-based Lazy.	7	0%	100%	0%	4	0%	100%	0%	4	25%	75%	0%

preferred the non-smelly pipe indicated that it makes the most sense to "consolidate similar modules" into a simpler structure.

The responses in Task 2 overwhelmingly favored the non-smelly pipe across all user groups. This is especially impressive when we consider the context of the pipe, in that it introduces abstraction using an extra module to parameterize a string constant. One notable exception is the two users who had low scores on their qualification tests. One of their responses expresses general confusion over the behavior of the pipe while the other was concerned about the increased number of wires introduced by the additional module.

In Task 3, the responses are varied among the different user groups. Among the end users who chose the non-smelly pipe, they all said that one *regex* module was easier to understand than two, and the degreed users tended to agree. On this particular task, it is interesting to also consider the average time to completion (O_5). In Table B.1, we see that the average time to completion for end users was nearly 4 times as long as that for degreed users, and Table B.2 shows similar results for the participants who had a low qualification score compared to participants who scored higher. We attribute this, in part, to the presence of a *regex* module, which may be a more familiar concept to users with computer science degrees or those who have a higher understanding of Yahoo! Pipes.

For Task 4, the responses were varied, but a majority of the participants (60%) were indifferent about which pipe would be easier to maintain. Just over one-fourth of the participants preferred the non-smelly pipe (27%). We note that the end users spent over twice as much time (O_5) to perform this task as the degreed users (see Table B.1), perhaps indicating that it took them longer to consider the implications of including invalid sources. The participants with low qualification scores generally favored the non-smelly pipe; participants who selected the non-smelly pipe noted that it's easier to update a pipe with fewer fields, since it is smaller. Across all groups, many of the participants who were neutral displayed a lack of understanding of what it would mean to focus on future updates (e.g., "both have the same basic structure and will be about as difficult to update," "Apart from the modules in A which are causing errors, there is no difference between the 2 pipes"). Yet other neutral participants desired more support and protection against errors that would come from broken sources (e.g., "Neither pipe seems to protect against deleted websites, so they seem equally update-friendly," "it seems that both would have the same issue with a deleted website").

In Task 5, a majority of the end users preferred the smelly pipe while a majority of the degreed users preferred the non-smelly pipe. One end user who preferred the smelly pipe indicated that while the modules are deprecated, they are still available, and the smelly pipe is easier to follow. This shows a clear misunderstanding of the dangers of using deprecated modules. Other participants shared similar sentiments about the smelly pipe, indicating that the loop introduced in the non-smelly pipe (to replace a deprecated module) makes it more difficult to understand. In general, the responses on this task displayed a focus on the understandability of the task and not necessarily the removal of deprecated sources as a service to others in the community who may be new to the environment and unfamiliar with the older, deprecated, and undocumented modules.

In Task 6, a majority of end users and half of the degreed users preferred the non-smelly pipe. Only participants with a low qualification score preferred the smelly pipe, but a scattering of participants across all groups were neutral about which pipe was easier to update. One of the neutral participants with a high qualification score noted that the pipes are the same to update, but the non-smelly pipe is the "most efficient." The high scoring participants who preferred the non-smelly pipe indicate that removing duplicates sources eases extra work that could come from the need to "track down multiple entries in your pipe output in the future," recognizing that duplicated sources lead to duplicated entries.

The responses for Task 7 tended to favor the smelly pipe across every user group. The general sentiments among participants who preferred the smelly pipe was that it was easier to understand because "you don't have to look at the subpipe to find out what it does," in other words, the behavior is not hidden by a subpipe module. We also note that the correlation between higher education (O_1) and the selection of the non-smelly pipe (O_3) is abnormally strong for this task, $r = -0.6434$ (see Table B.6), indicating that higher education correlates strongly with selection of the *smelly* pipe. This may imply that the end users will trust the functionality of a subpipe more than a degreed user, and thus prefer the non-smelly pipe. For example, one end user noted that the non-smelly pipe was easier to understand "since the subpipe behavior was described in text," exhibiting a general trust in the subpipe's behavior based on a textual description. We will further explore this task in light of an output task that also uses subpipes for abstraction in Section 3.3.5.

In Task 8, nearly all participants selected the non-smelly pipe instead of the smelly pipe. Most of the end user participants who favored the non-smelly pipe focused their responses on the omission of the inoperable module and the simplistic structure that resulted (five of the seven participants) instead of conformance to the canonical ordering set by the community (two of the seven participants). Similarly, only three of the seven degreed user participants cite the canonical ordering as the reason for their selection. We see similarly varied responses across the user groups based on qualification score, except that none of the respondents with low qualification score mention canonical ordering. Another interesting observation with this task is that there is an unusually high correlation between higher education (O_1) and the selection of the non-smelly pipe (O_3), $r = 0.8510$ (see the *Education & Non-Smelly* column in Table B.6), which may indicate that users with more computer science education are more likely to recognize the importance of coding standards to the community.

3.3.4.2 Experiment #2: Output Analysis and Correctness

Table 3.11 presents the correctness (O_4) of responses summarized across all tasks in the second experiment. The *Segment* column indicates the measure by which the population was segmented (i.e., based on education (O_1) or qualification score (O_2)), the *Group* column indicates the specific user group, and *Responses* indicates the total number of preference task responses received from a particular group. The $\%$ *Correct* column indicates how many participants selected the correct output and $\%$ *Incorrect* indicates how many participants selected an incorrect response. Participants were able to select the correct output in most cases (87%). Degreed users selected the correct output more often than end users, and participants with higher scores on the qualification test were able to select the correct answer more often than those with lower scores.

We further break down the user responses based on task, shown in Table 3.12. The *Type* column indicates the pipe structure, whether it is clean or smelly, indicating the control and experimental groups, respectively. The results of the output tasks segmented by education (O_1) are presented in Table 3.13 and the task results for the user groups segmented based on qualification scores (O_2) are shown in Table 3.14.

Table 3.11: Overall Correctness on Output Analysis Tasks

Segment	Group	Responses	% Correct	% Incorrect
O_1	End Users	16	75%	25%
	Degreed Users	14	100%	0%
O_2	High Score	14	93%	7%
	Middle Score	9	89%	11%
	Low Score	7	71%	29%
Overall	All Users	30	87%	13%

Table 3.12: Overall Correctness for Output Analysis Tasks by Treatment

Task	Type	Responses	% Correct	% Incorrect
9	Clean	8	88%	12%
	Smelly	6	100%	0%
10	Clean	6	83%	17%
	Smelly	10	80%	20%

Table 3.13: Correctness Results by Education (O_4 and O_1)

Task	Type	End Users			Degreed Users		
		#	% Correct	% Incorrect	#	% Correct	% Incorrect
9	Clean	6	83%	17%	2	100%	0%
	Smelly	1	100%	0%	5	100%	0%
10	Clean	4	75%	25%	2	100%	0%
	Smelly	5	60%	30%	5	100%	0%

Table 3.14: Correctness Results by Qualification Score (O_4 and O_2)

Task	Type	High Score			Middle Score			Low Score		
		#	% Correct	% Incorrect	#	% Correct	% Incorrect	#	% Correct	% Incorrect
9	Clean	2	50%	50%	4	100%	0%	2	100%	0%
	Smelly	5	100%	0%	0	0%	0%	1	100%	0%
10	Clean	4	100%	0%	1	100%	0%	1	0%	100%
	Smelly	3	100%	0%	4	75%	25%	3	67%	33%

Overall, we see that the degreed users were able to select the correct answer for the clean and the smelly pipes in all cases. For Task 10, the end users show a 15% increase in their ability to recognize the correctness of the clean pipes (control group) versus the smelly pipes (experimental group), and we see similar behavior (25% improvement) among the participants in the middle score qualification group. The experimental group was better able to understand the pipe in Task 9, as more participants were able to select the correct answer in the smelly pipe. Regardless, this data does show that pipes are understandable enough for users to select the correct answer from a list of potential outputs. Further, the transformations we performed on the dirty pipes to remove the smells and create the clean pipes do not negatively impact the understandability of the pipes generated by the community.

3.3.5 Summary

For 61% of the responses in the first experiment, the participants preferred the non-smelly pipes; for 18%, the participants were neutral about their preference. There was a general theme among all the user responses, that smaller pipes with simpler structures and fewer parameters are preferable. Additionally, users also recognized the utility of abstraction in making a pipe easier to maintain (see Task 2). We also saw that some pipes with more complicated constructs, such as Task 3 with the *regex* module, took end users a much longer time to analyze than those pipes with simpler constructs.

There is a notable exception to the preference toward smaller and more abstract pipes that was shown in Task 7, where the users found the subpipe to be less understandable. If we compare this to the correctness of responses in the control group of Task 9, which introduces three instances of a subpipe, we find that the subpipe does not impede a participants' ability to select the correct answer. Even though participants in Task 7 selected the pipe without a subpipe as the most understandable one, that does not mean they are incapable of understanding a pipe's behavior if it contains subpipe routines. We conjecture that the preference toward the less abstract pipe in Task 7 is the result of the pipe's simplistic structure, and users may prefer the subpipe abstraction for more complex pipes. Evaluating this conjecture would require further study and is left for future work.

Returning to the research question, *Are pipes with bad program characteristics (i.e., smells) less understandable or desirable than pipes without such characteristics?*, we can conclude that end users can certainly tell a difference between pipes with and without smells, and in general, preferred the non-smelly pipes to the smelly pipes. This theme was pervasive across most of the preference tasks. Although we did not notice common differences between user groups segmented based on education or qualification scores, we do note that these variables do not impede a participant's ability to analyze and understand pipes.

We also note several threats to validity of the conclusions drawn from this study. First, the high percentage of correct answers on the output tasks may have resulted from a correct answer that was too obvious or from pipes that are too trivial. We tried to control for this by including answers that have subtle differences, such as answers b and c in Task 9 and answers b and d in Task 10, and by selecting pipes that are reasonably large and complex, as indicated in Section 3.3.1.2. Second, the pipes selected here may not be completely representative of the smells as they are defined, which may limit the generalizability of the user responses with respect to the specific smells. We controlled for this by selecting pipes directly from the public repository of Yahoo! Pipes to ensure that the smells were not artificially inserted. Third, our measures may capture more events than we originally intended (e.g., the time to completion (O_5) on this tasks may include idle time away from the computer, a user could have set up multiple Mechanical Turk accounts to complete the same task multiple times, thus introducing learning effects). Last, only a small pool of participants, 22, was considered for this study, which may limit the generalizability of our findings. Further studies that include more participants to reduce this threat are left for future work.

3.4 Smells in the Community Artifacts

Motivated by end users' ability to recognize smelly pipes as being less desirable than non-smelly pipes, we analyzed a sample of pipes programs found in Yahoo!'s public repository and recorded how often each smell was present in those pipes. We found that each of the smells exists in at least 5% of our sample and that 81% of the sampled pipes contained at least one smell. In this section, we describe our sampling strategy and collection techniques used to gather artifacts from Yahoo's public repository, the smell detection infrastructure we built to analyze the pipes, and then present the results of an empirical study of 8,051 pipes programs extracted from the Yahoo! Pipes public, detailing the frequency of occurrence for each smell.

3.4.1 Artifact Selection and Collection

In order to detect smells in pipes (smell detection is described in Section 3.4.2), we had to collect pipes programs from Yahoo!'s public repository. However, the only available method for interacting with a pipes program is through the browser, since Yahoo! does not provide an API with which developers can interact with Yahoo! Pipes. To collect pipes programs for analysis, we wrote an automated scraper that will intercept HTTP traffic between a browser and the Yahoo! Pipes server and log the JSON representation of a pipe. Our scraper will obtain one pipe every ten seconds so as not to exceed the daily request limit imposed by Yahoo!'s servers.

Using our custom scraper, we obtained 10,362 pipes from Yahoo!'s public repository in approximately 30 hours. This number corresponds to the set of distinct pipes returned from 20 consecutive queries to the Yahoo! Pipes repository, where each query returns a maximum of 1,000 pipes. Each pipe in the query result must be loaded individually to obtain its JSON representation.

To obtain a pool of pipes that is representative without restricting the selection based on their configuration or structure (since that may affect the frequency of smells), we constrained the queries to pipes containing at least one of the 20 most popular data sources, as reported in January 2010. Among the pipes we collected, the average size is over 8 modules per pipe. We further trim the sample of trivial pipes and retain only those with at least four modules. The minimum size requirement was imposed to remove any pipes with trivial structures, since four modules is the minimal number necessary to create a pipe with multiple generator modules (and thus multiple paths to the output), as shown in Figure 3.8. This resulted in the final sample of 8,051 pipes.

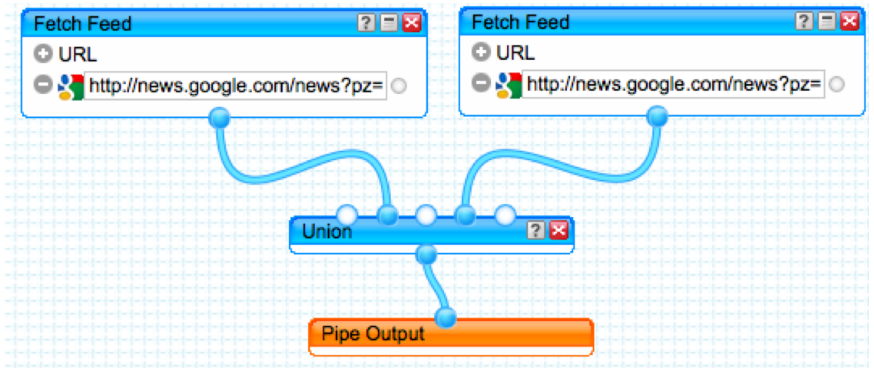

Fetch Feed
URL
http://news.google.com/news?pz=
Fetch Feed
URL
http://news.google.com/news?pz=
Union
Pipe Output

Figure 3.8: Four-Module Pipe with Two Generators

3.4.2 Smell Detection Infrastructure

We built a smell detection infrastructure that parses a JSON [16] representation of a Pipe and detects smells in the program. We parse the JSON representation using the standard Java parser provided by `json.org` [17], and detect the smells using our custom smell detection infrastructure, shown in Figure 3.9.

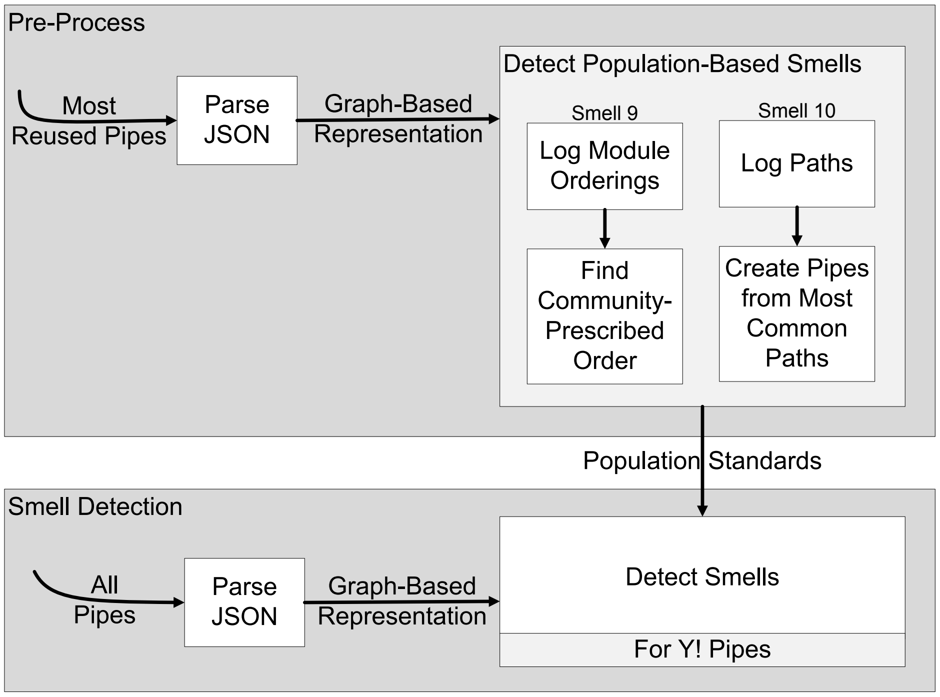


Figure 3.9: Smell Detection and Pre-Processing Infrastructure

Our smell detection infrastructure has two phases: *pre-processing*, and *smell-detection*. The *pre-processing* phase is used to detect the population-based smells. To do so, we consider only a sample of most reused pipes in the population under investigation, parse each pipe, create a graph-based representation (as described in Section 3.1), and then obtain copies of the pipe paths used by the population-based smells. Once each of the most reused pipes has been analyzed, we build population standards that are fed into the smell-detection part of the infrastructure.

Once the *pre-processing* is complete, the *smell-detection* phase begins. Here, we consider all pipes in the sample, parse the JSON representation to create a graph-based representation, and then crawl each pipe, logging instances of each smell defined in Section 3.2.

3.4.3 Smell Frequencies

For the population-based smells, we identify the most reused pipes as those that have been cloned more than 10 times, ($\sim$10% of the pipes in the sample). To detect Smell 9, we considered paths of length up to six, as this was the maximum length of read-only, order-independent operator modules in the population we considered. To detect Smell 10, we considered all those paths that were of length at least three and that existed in at least two of the most reused pipes in the population. Each smell presented here occurs in at least 5% of those pipes, and the frequencies of occurrence are summarized in Table 3.15.

Table 3.15: Most Prevalent Smells across 8051 Pipes

Smell Type	**Smell**	**Presence**
Laziness	Noisy Module	28%
	Unnecessary Module	13%
	Unnecessary Abstraction	12%
Redundancy	Duplicate Strings	32%
	Duplicate Module	23%
	Isomorphic Paths	7%
Environmental	Deprecated Module	18%
	Invalid Source	14%
Population-Based	Non-Conforming Operator Orderings	19%
	Global Isomorphic Paths	6%
All Smells		**81%**

While each smell individually impacts less than one-third of the pipes, we identified at least one smell in 81% of the pipes. In most cases, a single pipe is impacted by multiple smells, and each pipe contains instances of approximately two smells. This implies that the smells are not only common, but that the issues encoded in the pipes are non-trivial to detect and remove.

Chapter 4

Refactoring

To target and remove the most prevalent code smells, we have devised a set of semantics preserving pipe refactorings, defined in this section (see Appendix C for sketches of proofs for each refactoring). Similar to the smells, many of these refactorings have been inspired by refactorings proposed by Fowler [10]. We also present the adaptation of the implementation of the refactorings to fit the Yahoo! Pipes language, the infrastructure we built to automate the refactoring process for pipes, and the results of an empirical analysis where we apply the refactorings to the 8,051 pipes used in the smell detection analysis presented in Section 3.4. We conclude with a discussion of the generalizability of our techniques.

4.1 Refactoring Definitions

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 one of the smells defined in Section 3.2, and P_{after} is the refactoring postcondition. In our specification, we have further decomposed each refactoring into a set of more basic transformation rules utilizing the

$setter\_str(m)$	$m.type = setter.string$
$gen(m)$	$m.type = gen$
$union(m)$	$m.name = union$
$split(m)$	$m.name = split$
$in\_wire(m, w)$	$w.dest = m \wedge w.fld = \varnothing$
$field\_wire(m, w)$	$w.dest = m \wedge w.fld \neq \varnothing$
$out\_wire(m, w)$	$w.src = m$
$joined\_by(m_i, m_j, w)$	$out\_wire(m_i, w) \wedge in\_wire(m_j, w)$
$joined\_fld(m, f, w)$	$out\_wire(m, w) \wedge field\_wire(owner(f), w) \wedge w.fld = f$

Figure 4.1: Shorthand for Common Predicates Used in Refactoring Definitions

actions performed by pipe programmers (set, add, remove, move, copy) on pipe components (nodes, wires, and fields) and using visual depictions to complement the presentation of the most complex transformations. Figure 4.1 presents the shorthand notation used in the refactoring definitions.

4.1.1 Reduction

This category of refactorings focuses on removing unnecessary fields and modules that result from duplicated or lazy components. The overall result is a new pipe that is semantically equivalent to the original pipe, yet it is smaller in terms of fields, wires, or modules.

Ref 1. Clean Up Module: *removes any fields that are empty or duplicated within a module.*

$\boldsymbol{P_{before}}$	*Smell 1: Noisy Module*
Params	$Pipe$*, empty or duplicated field* f
Transf.	*set* $m = owner(f)$
	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).*

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 for the modules in the path to consume, are unnecessary and can be removed.*

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

Case 2.2. Lazy Module *that does not perform any operation or performs unnecessary redirection can be removed and its wires reconnected.*

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

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" [10].*

```
P_before   Smell 3: Unnecessary Abstraction
Params     $Pipe$, unnecessary module $m$
Transf.    String $s =$ “”
           for $k = 1 \cdots \mid m.\mathcal{F} \mid$
              append $m.\mathcal{F}[k]$ to $s$
              remove $m.\mathcal{F}[k]$
           $\forall w \in \mathcal{W} \mid out\_wire(m, w)$
              set $(w.fld).value = s$
              remove w
           remove $m$
P_after    $m \notin Pipe$,
           $\forall w \in Pipe.\mathcal{W} \mid out\_wire(m, w), (w.fld).value = s$ & $w \notin Pipe$
```

4.1.2 Consolidation

The goal of these refactorings is to unify duplicated code in an effort to simplify pipe structures and reduce their sizes, as was a desirable pipe characteristic expressed by end users in our study, as discussed in Section 3.3.4. To this end, these refactorings merge operator modules performing transformations that could be accomplished with just one module and collapse duplicate paths that perform identical transformations on separate lists of items, which are later merged.

Ref 4. Merge Redundant Modules: *merges operators and path-altering modules that are connected or perform the same operation along the same path.*

Case 4.1. Merge Consecutive Operators *with the same name as they can be consolidated into one module, decreasing the size and complexity of the pipe. This refactoring is motivated by the* Inline Class *refactoring that moves all the features of one class into another class, and then deletes it [10]. Here, m_j is being inlined, and m_i absorbs all its features. See Figure 4.2 for an example.*

P_{before} *Smell 5.1: Consecutive order-independent operators*

Params *Pipe, order independent operators m_i, m_j*

Transf.

```
$\exists w_i, w_j \in \mathcal{W} \mid joined\_by(m_i, m_j, w_i) \wedge out\_wire(m_j, w_j)$
    set $w_i.dest = w_j.dest$
for $k = 1 \ldots m_j.\mathcal{F}$
    append $m_j.\mathcal{F}[k]$ to $m_i.\mathcal{F}$
remove $m_j, w_j$
```

P_{after} $m_j, w_j \notin Pipe, m_i.\mathcal{F} = m_i.\mathcal{F}_{old} \cup m_j.\mathcal{F}$

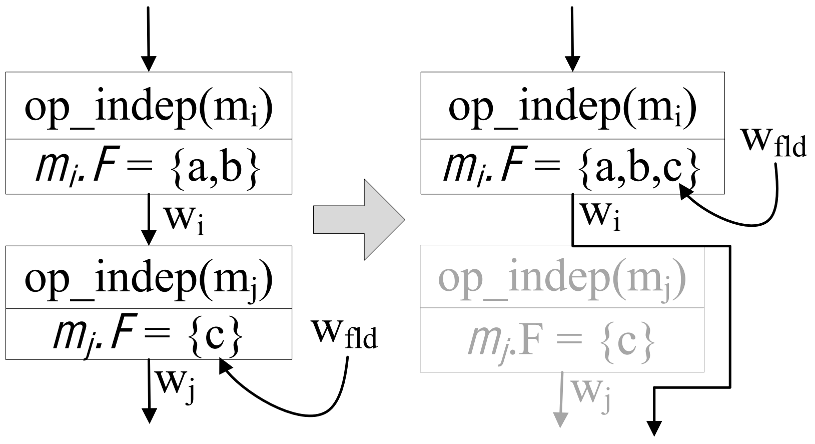


Figure 4.2: Merge Consecutive Operators

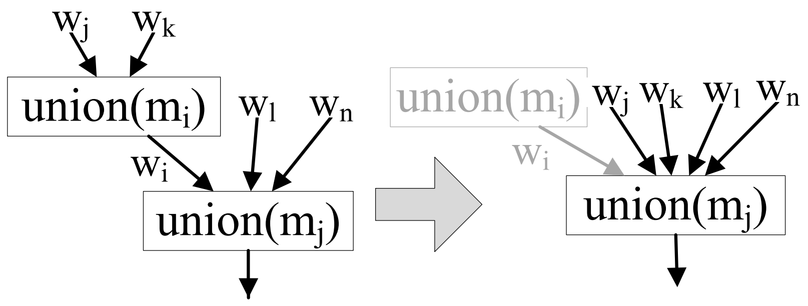


Figure 4.3: Merge Path Altering

Case 4.2. Merge Path Altering Modules *that are underutilized by consolidating the incoming or outgoing wires. This refactoring is also motivated by* Inline Class *[10]. See Figure 4.3 for an example of refactoring consecutive union modules.*

$\boldsymbol{P_{before}}$ *Smell 5.2: Consecutive path-altering modules*

Params $Pipe$*, path altering modules* m_i, m_j

Transf *If* $union(m_i)$, $\exists w_i \in \mathcal{W} \mid joined\_by(m_i, m_j, w_i)$

$\forall w \in \mathcal{W} \mid in\_wire(m_i, w)$, ***set*** $w.dest = m_j$

If $split(m_i)$, $\exists w_i \in \mathcal{W} \mid joined\_by(m_j, m_i, w_i)$

$\forall w \in \mathcal{W} \mid out\_wire(m_i, w)$, ***set*** $w.src = m_j$

remove m_i, w_i

$\boldsymbol{P_{after}}$ $m_i, w_i \notin Pipe$

Case 4.3. Merge Subsequent Operators *with the same name and parameters as they are redundant. In the precondition, we guarantee that module m_j postdominates module m_i, and since both modules perform the same operation on the data, the postdominated module m_i is unnecessary and can be removed. See Figure 4.4 for an example.*

P_{before} *Smell 5.4: Identical subsequent operators*

Params *$Pipe$, identical operator to be removed m_i*

Transf. $\exists w_i, w_j \in \mathcal{W} \mid in\_wire(m_i, w_i), out\_wire(m_i, w_j)$

set $w_i.dest = w_j.dest$

$\forall w \in \mathcal{W} \mid field\_wire(m_i, w)$

remove w

remove m_i, w_j

P_{after} $m_i, w_j \notin Pipe$

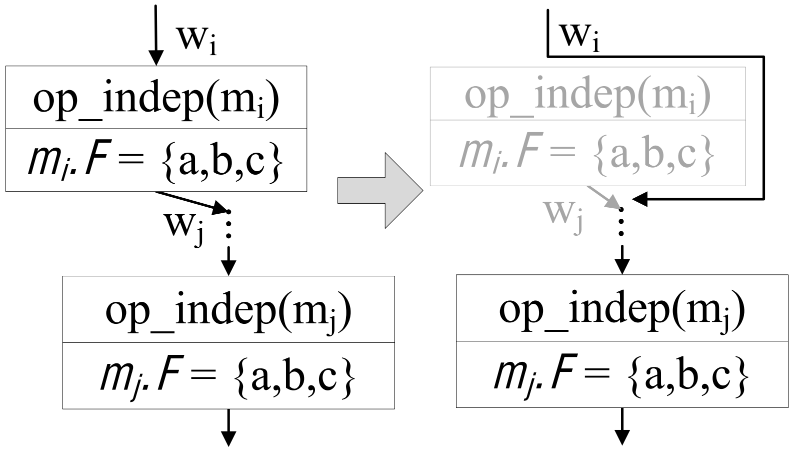


Figure 4.4: Merge Subsequent Operator

Ref 5. Collapse Duplicate Paths: *paths that are aggregated using the same union module can often be consolidated into a single path, simplifying the pipe structure. This can occur in the presence or absence of operator modules along the paths, so we present two cases for clarity.*

Case 5.1. Joined Generators *whose output is unmodified prior to aggregation, reducing the complexity and size of the pipe. See Figure 4.5 for an example.*

$\boldsymbol{P_{before}}$	*Smell 5.3: Joined generators*
Params	$Pipe$, *joined generator modules* m_i, m_j
Transf.	*for* $k = 1 \cdots \mid m_i.\mathcal{F} \mid$
	append $m_i.\mathcal{F}[k]$ *to* $m_j.\mathcal{F}$
	$\exists w_i \in \mathcal{W} \mid out\_wire(m_i, w_i)$
	remove w_i
	remove m_i
$\boldsymbol{P_{after}}$	$m_i, w_i \notin$ *Pipe*, $m_j.\mathcal{F} = m_j.\mathcal{F} \cup m_i.\mathcal{F}$

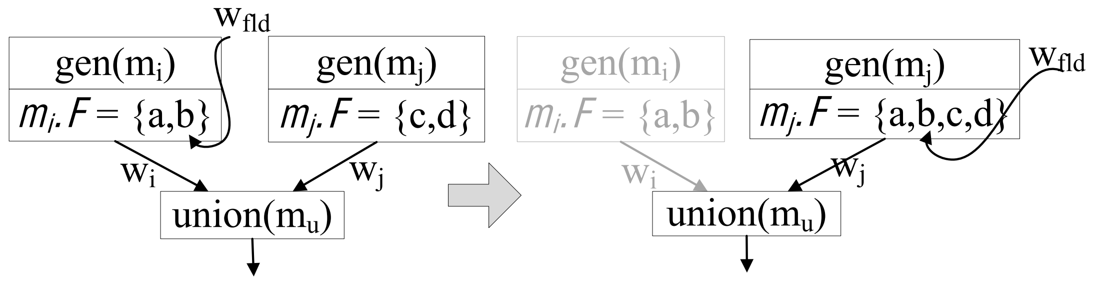


Figure 4.5: Joined Generators

Case 5.2. Identical Parallel Operator Pairs *that perform the same operation prior to aggregation can be collapsed into one path by merging the generator modules. An instance of this refactoring is illustrated in Figure 4.6 for two generator modules.*

```
P_before  Smell 5.5: Identical parallel operators
Params    Pipe, operators $m_i, m_j$, and modules $m_k, m_l$
Transf.   $\exists w_i, w_j, w_k, w_l \in \mathcal{W}, \exists m_u \in Pipe \mid$
            $joined\_by(m_i, m_u, w_i), joined\_by(m_j, m_u, w_j),$
            $joined\_by(m_k, m_i, w_k), joined\_by(m_l, m_j, w_l)$
            if $gen(m_k) \wedge gen(m_l)$
              for $x = 1 \cdots \mid m_k.\mathcal{F} \mid$
                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 $union(m_k) \wedge union(m_l)$
              $\forall w \in \mathcal{W} \mid in\_wire(m_k, w)$
                set $w.dest = m_l$
              remove $m_k, w_k$
            remove $m_i, w_i$
P_after   $m_i, m_k, w_i, w_k \notin Pipe$
```

4.1.3 Abstraction

These refactoring focus on abstracting areas of the pipe in which there are duplicate fields and paths. The inspiration for these transformations draws from refactorings that aim to more cleanly compose and package code by abstracting common code into its own method or replacing an algorithm with a cleaner one [10].

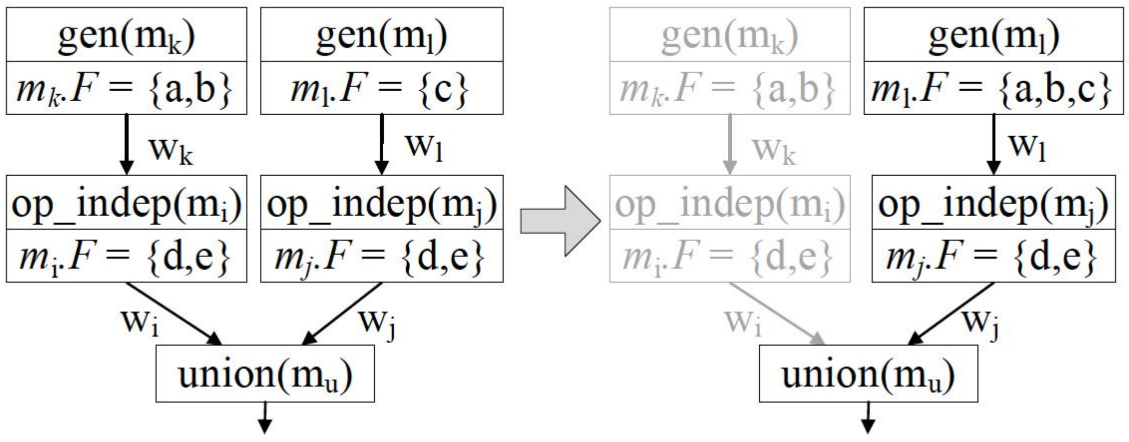


Figure 4.6: Identical Parallel Operators (two generators)

Ref 6. Pull Up Module: *extracts duplicate strings into a separate module and provides the values via wires to the previous owners of the duplicated strings. This refactoring is analogous to the common* Extract Method *refactoring that abstracts the same expression from two methods in the same class into its own method, and then invokes the new method from the original expression locations [10].*

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* $Pipe.\mathcal{W} \mid joined\_fld(m, f_j, w_j)$

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. 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 [10]. In this case, we replace all instances of the path with a cleaner module. For example, in Figure 1.3, a subpipe was created to replace two paths from Figure 1.2, from A to E, and from C to G. The field values from A, C, and G were copied to their respective subpipes. The wire providing the field value to E was reconnected to the field from E in subpipe $A + E$.*

```
$P_{before}$   Smell 6: Isomorphic Paths
Params         $Pipe$, isomorphic paths $p$ and $p'$
Transf.        % Build subpipe
  (1)          create pipe $newPipe$
               add module $o$ to $newPipe.\mathcal{M} \mid o.name = output$
               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 $newPipe.\mathcal{M} \mid q.type = setter.user$
  (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 $Pipe.\mathcal{M} \mid r.name = subpipe(newPipe)$
                  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 $(f.value! =``")$, copy $f.value$ to $r.q.value$
                  remove $a$
$P_{after}$    $p$ and $p' \notin Pipe$, $\exists_2 subpipe(newPipe) \in Pipe$
```

4.1.4 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, similar in spirit to previous work that used refactorings to update references to deprecated library classes in Java programs [2].

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}.*

$\boldsymbol{P_{before}}$ *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 $m_{dep}$
```

$\boldsymbol{P_{after}}$ $m_{dep} \notin Pipe$

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.*

$\boldsymbol{P_{before}}$ *Smell 8: Invalid Sources*

Params $Pipe$, *field* f *referring to* $es \in ExternalSources$

Transf. $m = owner(f)$

remove f *from* m

$\boldsymbol{P_{after}}$ $f \notin m.\mathcal{F}$

4.1.5 Population-Based Standardizations

These refactorings exploit the availability of a large repository of pipes developed by end users to standardize the practices across the community in order to facilitate reuse. The inspiration for these transformations draws from the vast amount of community knowledge encoded in repositories of programs and our ability to extract common knowledge and standards to enrich the programs created by a single user.

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. The results of HIT 8 in Table 3.8 in which users overwhelmingly showed that the pipe with the canonical module ordering is easier for others to understand, motivates this refactoring.*

$\boldsymbol{P_{before}}$ *Smell 9: Non-conforming module orderings*

Params $Pipe$*, non-conforming path* p*, prescribed path* $ppres$

Transf.
```
add path $ppres$ to $Pipe$
$w_i \in \mathcal{W} \mid in\_wire(p(first), w_i)$
    set $w_i.dest = ppres(first)$
$w_j \in \mathcal{W} \mid out\_wire(p(last), w_j)$
    set $w_j.src = ppres(last)$
remove $p$
```

$\boldsymbol{P_{after}}$ $ppres$ *in place of* p

Ref 11. Extract Global Subpipe: *this is the generalization of Refactoring 7 to operate across a population of pipes. The difference with this refactoring is in the broadening of the space on which the pattern identification occurs, that is, across multiple pipes as opposed to across multiple paths within the same pipe. 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($a = p \in Paths$*),* $newPipe = getSubPipe(a)$
$\boldsymbol{P_{after}}$	$\forall p \in Paths \mid p \notin Pipes,$
	$\exists_1 subpipe(newPipe) \in Pipe$

4.2 Refactorings Adapted to Yahoo! Pipes

This section describes the additional refactoring constraints and adaptations we performed to fit the Yahoo! Pipes language. We discuss the impact of these changes in Section 4.4. For the population-based refactorings, as was done for the smell detection study described in Section 3.4.3, we identify the most reused pipes as those that have been cloned more than 10 times, ($\sim$10% of the pipes in the sample).

Refactoring 1: Clean Up Sources. In the case of duplicate fields in the precondition, this refactoring was performed on the generator modules (the most common case) but not on operator modules which would have required an additional set of dependency analyses tailored to a large variety of operators.

Refactoring 3: Push Down Module. The refactoring of the $urlbuilder$ module required additional processing to insert separator symbols when assembling a url string using its fields (e.g., base url, paths, query parameters).

Refactoring 4.1: Merge Consecutive Operators. This refactoring was only performed for operator modules that can accommodate multiple fields (e.g., $sort$, $filter$, $regex$, $rename$). Operator modules, such as $loop$ and $unique$, that do not support a variable number of fields, cannot be merged. Also, for operators with non-wireable fields, additional matching constraints were added. For example, the $filter$ module contains two non-wireable fields whose values are set using a selectable drop-down menu. To merge consecutive $filter$ modules, we require these values to match.

Refactoring 4.2: Merge Path Altering and **Refactoring 5.2: Identical Parallel Operator Pairs.** Path-altering modules in Yahoo! Pipes have a bounded number of incoming and outgoing 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; we used this information as a guide for refactoring. The following deprecated modules are replaced: $foreach$, $foreachannotate$, $contentanalysis$, and $babelfish$.

Refactoring 9: Remove Deprecated Sources. When accessing information from ExternalSources, error codes such as 404 Not Found or 503 Service Unavailable may be returned. We use this information to remove deprecated sources. This refactoring is applied to generator and string-setter modules, but not to user-setter modules because they can be overwritten at run-time.

Refactoring 10: Normalize Order of Operations. We generate $PPres$ by considering the most reused pipes and identifying paths of size two to five, containing read-only and order-independent modules that appear in multiple pipes within this subset of pipes.

Refactoring 11: Extract Global Subpipe. We generate $PGPaths$ by considering the most reused pipes and identifying paths of at least length three that appear in multiple pipes within the subset.

4.3 Refactoring Study Infrastructure

To facilitate an assessment of the effectiveness of the refactorings, we built a manipulation infrastructure, shown in Figure 4.7.

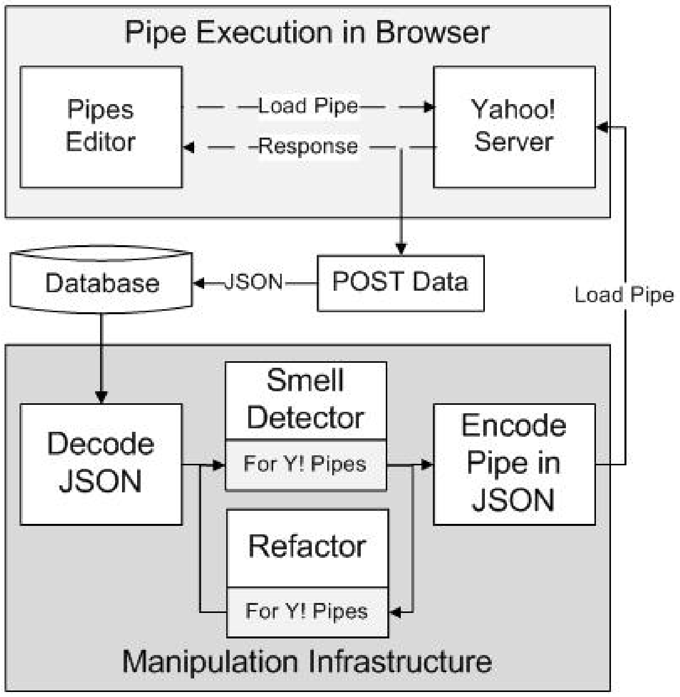


Figure 4.7: Manipulation Infrastructure

Figure 4.7 illustrates the pipe collection process described in Section 3.4.1, contains the pre-processing and smell detection infrastructure for all 10 smells described in Section 3.4.2 (*Smell Detector* component), provides transformations for all 11 refactorings subject to the language constraints described in Section 4.2, and supports the full grammar of Yahoo! Pipes. 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 [16] representation of the Pipe in the POST data. We stored the results in a database. These pipes are fed into the *Decode JSON* component (same as the *Parse JSON* step in Figure 3.9), which produces a graph-based representation of a pipe. The smell detection infrastructure indicates the smelly parts of the pipe, which are then targeted by the refactorings. Once a pipe has been sufficiently refactored, we can send an encoding of the refactored pipes to Yahoo!'s servers so the newly-updated version will be loaded in the Pipes Editor.

As part of the *Refactor* component in 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 helps us explore how refactorings may interact when applied in sequences (similar to what was described in Section 1.1). Figure 4.8 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.

```
**Require:** Pipe $\mathcal{PG} = (\mathcal{M}, \mathcal{W}, \mathcal{F}, owner)$
  $Map < Smell, Refactoring > smellRefMap$
**Ensure:** returns $\mathcal{PG}'$, a pipe with minimal smells
  $Set < Refactoring > ref$
  $Set < Smell > currentSmells, previousSmells$
  $\mathcal{PG}' = \mathcal{PG}$
  $currentSmells = detectSmells(\mathcal{PG}')$
  $previousSmells = \varnothing$
  **while** $previousSmells \mathrel{!=} currentSmells$ **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 $\mathcal{PG}'$
```

Figure 4.8: Greedy Algorithm

4.4 Effectiveness of Refactorings

Analyzing the pipes with the manipulation infrastructure yielded some promising results for the effectiveness of the refactorings at removing smells. For each smell, Table 4.1 presents the 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%. The final row, *Greedy Approach*, targets all smells, using the algorithm in Figure 4.8.

Table 4.1: Smells and Refactoring Effectiveness

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%								
	Push Down 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%

(Results are only reported for smell changes per pipe that are $\geq 5\%$.)

Seven of the refactorings applied individually are able to target and 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*. One other refactoring, *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, a module that is inoperative). This creates an opportunity for other refactorings, such as *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. The Noisy Module smell is particularly interesting in that only one refactoring targets this smell, Ref 1: Clean Up Module, yet three refactorings have a valuable impact. Further, the maximal impact of any refactoring on this smell is 18%, but when applied together, the total reduction is closer to 43%, illustrating how the application of multiple refactorings may be necessary to more completely address a smell.

We explore the effect of applying a sequence of refactorings utilizing the algorithm in Figure 4.8. The results, shown in the last row of Table 4.1, 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 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 since those values can be over-written by a user at runtime.

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. This means that the refactorings were able to completely eliminate the smells in nearly 80% of the pipes that had smells to begin with. In addition, the average number of smells per pipe was reduced from eight to one through the proposed refactorings.

4.5 Generalizability and Threats

There are many web mashup environments available to end users, as discussed in Section 2.2, though our studies focus on just one of those environments, Yahoo! Pipes. This environment was selected to maximize the potential impact of the findings (given the popularity of Yahoo! Pipes), and because of the availability of a rich public repository to support a large study on smell detection and refactorings. In this section, we begin to address this threat by assessing the generalizability of our approach and performing a manual inspection and analysis of the pipes available in the newer DERI Pipes repository. Then, we discuss two more threats to validity.

4.5.1 Generalizability

Of the 110 published DERI pipes, 58 meet the size selection criteria used for our Yahoo! Pipes study. In spite of the small 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, with an average of 1.4 total smells per pipe.

We note, however, that particular DERI language constructs and constraints will require further tailoring of our infrastructure. For example, there may never be an instance of *Smell 5.2: Consecutive path-altering modules* because there is no limit on the number of incoming or outgoing wires for the path-altering modules. This smell was particularly common in Yahoo! Pipes because of the limit of five incoming wires on the *union* module. Similarly, language constructs may also impede certain refactorings. Since DERI's generator modules do not support multiple fields, they cannot be merged, so the *Smell 5: Duplicate Module*, which affects 27% of the pipes, cannot be detected and thus *Ref 1:Clean Up Module* cannot be applied.

Despite the language limitations, we observe that three out of the five smells we detected can be successfully refactored. *Smell 6: Isomorphic Paths* impacts 14% of the pipes and can be eliminated using *Refactoring 7: Extract Local Subpipe*. *Smell 8: Invalid Source* impacts 10% of the pipes and can be eliminated using *Refactoring 9: Remove Deprecated Sources*. Additionally, 9% of the pipes contain unnecessary modules, which can be removed like they are in *Refactoring 2: Remove Non-Contributing Modules*.

4.5.2 Other Threats to Validity

The first threat regarding the generalizability of our approach was addressed in part through the analysis of the DERI repository, where we discovered that many of our smells and refactorings could be applied directly. Still, it remains to be explored whether the smells and refactorings will be relevant in other environments.

A second threat to the general validity of our results that we will address in future work is the assessment of the proposed refactorings in the hands of end users. Although the defined smells are prevalent across the pipes shared by the community members, the refactorings ultimately need to be assessed with end users to determine, for example, whether and how they are adopted in practice. We addressed this threat in part by our formative user study that showed end users' general preference toward pipes that lacked the smells we have identified.

A third threat concerns the correctness of the tools we have developed, which includes the smell detector and the refactorings, but also the components to, for example, scrape, decode, and load a pipe. In addition to the individual tests of each of those components, we have manually inspected the end-to-end transformations of over 200 pipes to increase our confidence in the tools.

Chapter 5

Conclusions and Future Work

End users are developing and sharing mashups in increasing numbers. However, a popular kind of mashup being created by end users, pipes, have many deficiencies such as being bloated with unnecessary modules, accessing broken sources of data, using atypical constructs, or requiring changes in multiple places even for minor adjustments because of the lack of abstraction. In this work, we have formalized the definitions for a family of code smells that identify these deficiencies and shown the value of the smells in two ways. First, we show that end users demonstrate a preference toward pipes that are smaller and lack our identified code smells. Second, we show that these code smells are pervasive in the community artifacts, impacting 81% of our large sample of 8,051 pipes developed by thousands of end users.

Inspired by how refactoring can benefit professional developers by targeting and removing smells, we have developed refactorings that target the most prevalent smells identified in the 8051 pipes. The refactorings include some adapted from more traditional programming domains (e.g., removal of unnecessary modules, pulling up and pushing down modules), but also some that are unique to the mashup domain, such as the population-based and consolidation 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 smells per pipe by almost 90%. Given these promising results, we envision several avenues for future work.

First, we want to study how end users can utilize the refactorings, and have made steps in this direction by developing a refactoring prototype. A user study will reveal how intuitive the refactorings are for end users and how likely an end user is to utilize a refactoring tool. Such a tool could also serve as a design critic that could perform context-sensitive suggestions during mashup development toward the goal of making the pipe more usable by others, or have application toward computer science education by helping students, especially those just learning to program, to identify and remove code smells in an effort to instill good programming practices.

Second, we want to broaden the family of refactorings to address other smells we have observed. For example, some urls return a 403 Forbidden or 401 Unauthorized response code when accessed. This affects urls in 3% of the pipes, but a forbidden URL is often the result of missing login credentials, and has implications for the correctness of the pipe when shared with others - considering that 66% of the pipes with forbidden urls had been cloned, addressing this error is important. We would also like to extend the population-based refactorings to better leverage the community resources and provide better user support. For example, we would like to extend the global isomorphic paths refactoring to scrape the Yahoo! Pipes repository and replace isomorphic paths with existing pipes from the community instead of creating new ones for the purpose of abstraction. With the work we have presented, we believe that we are just beginning to tap the power of such community based resources to assist end users in the development of mashups.

Appendix A

Mechanical Turk Implementation

A.1 Qualification Quiz for Web Mashup Understanding

Please Note: You must be at least 19 years old to participate. Completing this qualification exam involves four parts:

1. Answer questions about your background and programming experience
2. View tutorial information about the Yahoo! Pipes mashup environment
3. Answer comprehension questions about Yahoo! Pipes to evaluate your understanding of the environment
4. Read the Informed Consent notice

Only your answers to the questions in Part 3 will be graded for this exam. To pass the exam, you must answer at least 4/8 questions correctly in Part 3 and accept the Informed Consent form in Part 4. It is necessary that you complete Part 1, but we have no expectations about your answers.

A.1.1 Part 1: Background Questions

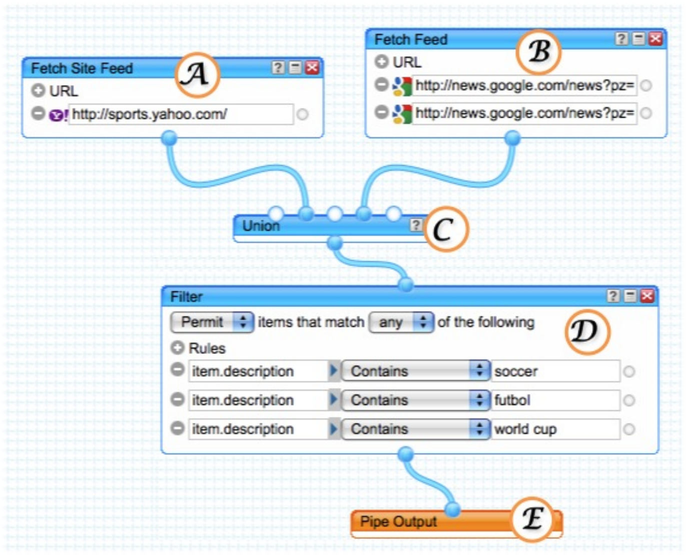
1. Are you at least 19 years old?
    a) Yes
    b) No
2. What is your gender?
    a) Male
    b) Female
    c) Prefer not to answer
3. What is your programming experience?
    a) Limited or No Experience
    b) Self-taught only
    c) On-the-job training only
    d) One or more classes, in high school or college, in computer science or related field
    e) Undergraduate/Graduate degree, in progress or completed, in computer science or related field
4. How long have you been programming (e.g., using languages like Java, C/C++, JavaScript, Perl, Python, etc.)?
    a) no experience
    b) less than 1 year
    c) 1–5 years
    d) 5+ years

A.1.2 Part 2: Tutorial Information

Please view the following tutorial video (http://video.yahoo.com/watch/5260536/13878389) on Yahoo! Pipes to prepare you to answer the comprehensive questions in Part 3. If you are already familiar with the Yahoo! Pipes environment, you may skip straight to the questions.

A.1.3 Part 3: Comprehensive Questions

Answer questions 1-8 below. Questions 1-6 are based on the screen shot of a pipe displayed below. Each module in the image is annotated with a letter for easy reference. Questions 7-8 are based on the Yahoo! Pipes environment. Feel free to consult the Yahoo! documentation as needed. (For the purposes of the appendix, the correct answers are indicated in *italics*.)



URL
http://sports.yahoo.com/

URL
http://news.google.com/news?pz=
http://news.google.com/news?pz=

Permit items that match any of the following
Rules
item.description Contains soccer
item.description Contains futbol
item.description Contains world cup

Figure A.1: Pipe for Qualification Test Questions

1. What is the behavior of Module B?
   a) Acts as the final output module of the pipe, aggregating data from all modules connected to it.
   b) Finds the locations of each item in the feeds specified by the websites and places them on a map.
   c) *Gathers the content from each website specified in the input fields.*
2. What is the behavior of Module C?
   a) *It aggregates the content from Module A and Module B.*
   b) It gathers the content from Module A or Module B, depending on which module works faster.
   c) It iterates through content from Module A, Module B, and Module D, only allowing unique items to pass through.
3. What happens to the content that passes through Module D?
   a) It goes to Module C.
   b) *It goes to Module E.*
   c) It doesn't go anywhere.
4. Let's assume that the website specified in Module A contains three items and each of the two websites in Module B contains four items. Two items in Module A have the word soccer in the title, but only one has the word soccer in the description. Five items from Module B have the word futbol in the description, and three of the five also have the word soccer in the description. There are no instances of the phrase world cup in any of the items. What is the maximum number of items that will reach Module E?
   a) Zero, since D blocks all items.
   b) One, since only one has the word soccer in the title.
   c) Three, since only three have both futbol and soccer in the description.

   d) *Six, since futbol or soccer appears in the description of six items.*
   e) Eleven, since Module D allows all items to pass through.

5. What happens in the absence of Module E?
   a) An error occurs and Yahoo! deletes the pipe.
   b) The output of the pipe is printed to the screen.
   c) *Nothing happens; there is no output.*

6. What happens if the link between Module C and Module D is removed?
   a) An error occurs and Yahoo! deletes the pipe.
   b) The output of the pipe is printed to the screen.
   c) *Nothing happens; there is no output.*

7. What does it mean to clone a pipe? (The tutorial video in Part 2 mentions cloning toward the end.)
   a) Copy a module from another pipe for your own use.
   b) *Copy an entire pipe for your own use.*
   c) Copy an entire pipe and insert it as a module in your own pipe.

8. The Yahoo! Pipes environment supports re-use of pipes. That is, a pipe can be re-used as a building block and inserted as a module in a different pipe. We refer to these as subpipes. Select the answer below that most closely fits the definition of a subpipe. (The following document may help you answer this question: http://pipes.yahoo.com/pipes/docs?doc=modules)
   a) Given Pipes A and B, A is a subpipe of B if the output of A is a subset of the output of B.
   b) A grouping of some, but not all, of the modules in a given pipe.
   c) *An entire Pipe that has been inserted as a module in a different pipe.*

A.1.4 Part 4: Informed Consent (IRB#20100410792 EX)

In order to pass this qualification exam, you must accept to the informed consent information, presented here:

–Purpose of the Research–

The goal of this study is to analyze the impact of coding practices in the maintainability and understandability of pipe-like web mashups in the context of the Yahoo! Pipes environment.

–Procedures–

You must be at least 19 years of age to participate in this study. Participation in this study will involve the completion of one or more HITs that will ask you to analyze pipes created in the Yahoo! Pipes environment. You will be asked multiple-choice and/or open-ended questions about the pipes. This study will be conducted from your personal computer via the Mechanical Turk website. Each HIT should take no more than 5-10 minutes to complete, though the allotted time by Mechanical Turk allows 60 minutes per HIT. To complete all 10 of the HITs available to you should take no longer than one hour in total.

–Risks and/or Discomforts–

Potential discomforts may come from the effort it takes to understand Pipes and complete the tasks.

–Benefits–

As a participant, you will be required to learn about Yahoo! Pipes, which could help you to streamline your activities on the Internet. Additionally, for each HIT completed (up to 10), you will receive compensation of $0.08, forapotentialtotalcompensationof$0.80. If you choose not to complete any HITs, you will not receive any monetary compensation.

–Confidentiality–

Your answers will be strictly confidential and will not be connected to your name, email, IP address, or any other identifying information.

–Opportunity to Ask Question–

You may ask any questions concerning this research and have those questions answered before agreeing to participate in or during the study. The e-mail address of the primary investigator is kstolee 'at' cse.unl.edu, and the e-mail address of the secondary investigator is elbaum 'at' cse.unl.edu. If you have questions concerning your rights as a research subject that have not been answered by the investigators or to report any concerns about the study, you many contact the University of Nebraska-Lincoln Institutional Review Board, telephone (402) 472-6965.

–Freedom to Withdraw–

You are free to decide not to participate in this study or to withdraw at any time without adversely affecting your relationship with the investigators or the University of Nebraska. Your decision will not result in any loss or benefits to which you are otherwise entitled, but compensation is only provided once HITs have been completed.

–Consent, Right to Receive a Copy–

Your acceptance certifies that you have decided to participate having read and understood the information presented. You may save this page for your records.

–Affiliations–

We are researchers in the ESQuaReD lab at the University of Nebraska-Lincoln and are in no way affiliated with Yahoo!.

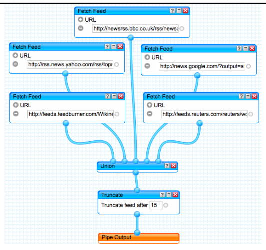
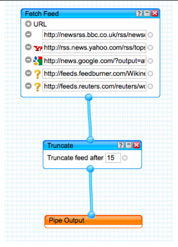
1. Do you agree to the information presented in the informed consent?
   a) Yes, I have read, understood, and accept the informed consent.
   b) No, I do not accept the informed consent.

A.2 Human Intelligence Tasks

A.2.1 Preference

A HIT in this category has a structure like the one shown here. The points of variability are pipe images (i.e., A and B), number of clones (i.e., X and Y), the context (i.e., Pipes with different structures can generate the same output.) and the characteristic being evaluated with the questions (i.e., to understand). Each preference HIT description that follows in Sections 1–8 defines these points of variability. We indicate the *smelly* and *clean* pipes for reference, but that information was not available to the users.

Table A.1: Preference HIT Example

A	**B**
 Fetch Feed URL http://newsrss.bbc.co.uk/rss/newsc http://rss.news.yahoo.com/rss/topi http://news.google.com/?output=r http://feeds.feedburner.com/Wikin http://feeds.reuters.com/reuters/w Union Truncate Truncate feed after 15 Pipe Output	 Fetch Feed URL http://newsrss.bbc.co.uk/rss/newsc http://rss.news.yahoo.com/rss/topi http://news.google.com/?output=r http://feeds.feedburner.com/Wikin http://feeds.reuters.com/reuters/w Truncate Truncate feed after 15 Pipe Output
X clones	Y clones

Click each image to open a larger view.

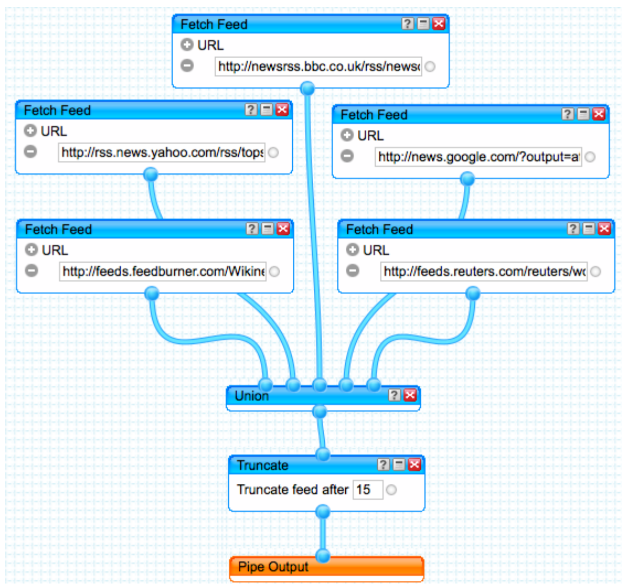
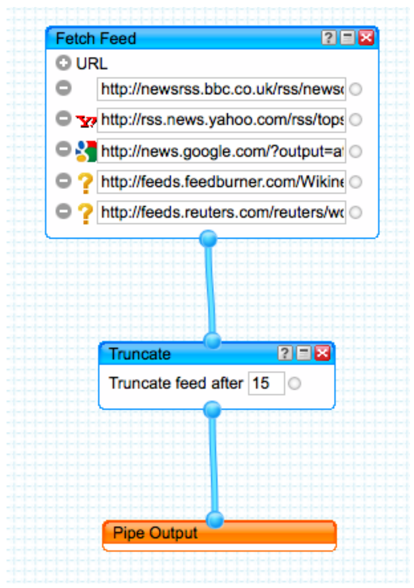
Take some time to understand the behavior of each pipe. To answer questions 1 and 2 below, consider the following context:

Pipes with different structures can generate the same output.

1. Select the pipe that is easiest **to understand**
   a) A
   b) B
   c) Same
2. Justify your answer (you must use at least 10 words in your explanation):

HIT 1. Preference Joined Generators

Table A.2: Joined Generators Smell and Joined Generators Refactoring

A (smelly)	B (clean)
	
65 clones	65 clones

Fetch Feed
URL
http://newsrss.bbc.co.uk/rss/news

Fetch Feed
URL
http://rss.news.yahoo.com/rss/tops

Fetch Feed
URL
http://news.google.com/?output=a

Fetch Feed
URL
http://feeds.feedburner.com/Wikine

Fetch Feed
URL
http://feeds.reuters.com/reuters/wo

Union

Truncate
Truncate feed after 15

Pipe Output

Fetch Feed
URL
http://newsrss.bbc.co.uk/rss/news
http://rss.news.yahoo.com/rss/tops
http://news.google.com/?output=a
http://feeds.feedburner.com/Wikine
http://feeds.reuters.com/reuters/wo

Truncate
Truncate feed after 15

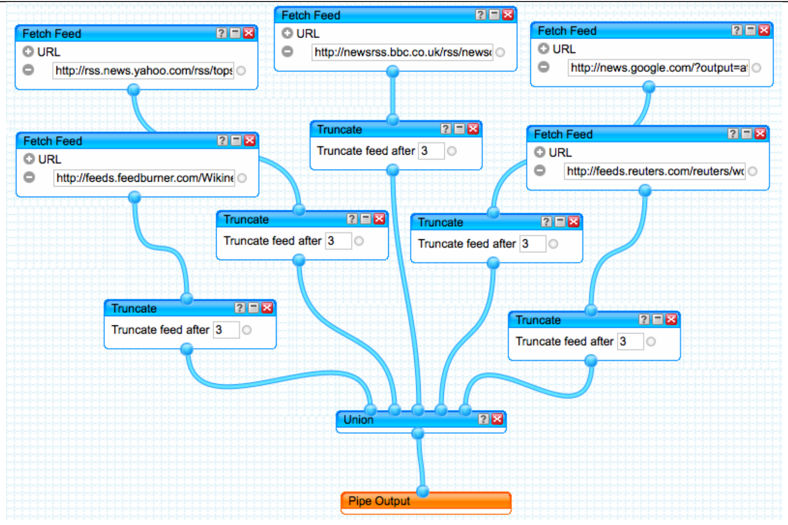
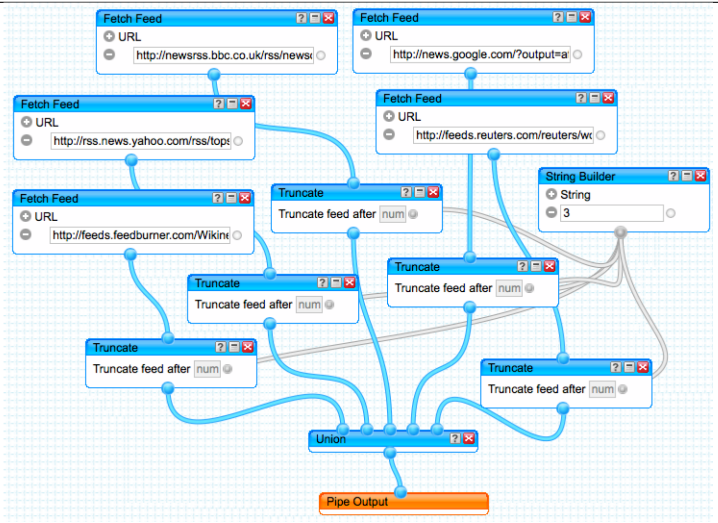
Pipe Output

Context: *Pipes with different structures can generate the same output, as is the case with Pipes A and B.*

Characteristic: *to understand*

HIT 2. Preference Duplicate Strings

Table A.3: Duplicate Strings Smell and Pull Up Module Refactoring

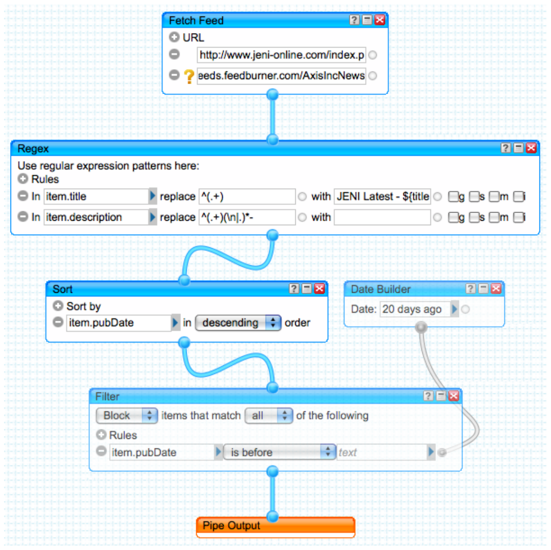
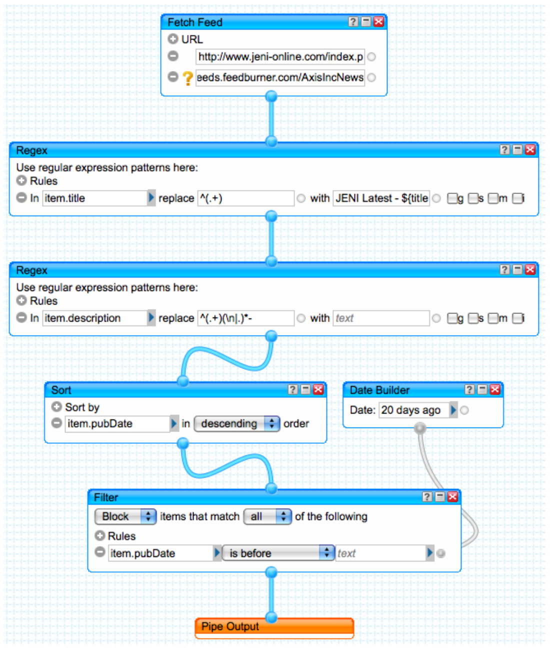
A (smelly)
 Fetch Feed URL http://rss.news.yahoo.com/rss/tops Fetch Feed URL http://newsrss.bbc.co.uk/rss/news Fetch Feed URL http://news.google.com/?output=a Fetch Feed URL http://feeds.feedburner.com/Wikine Truncate Truncate feed after 3 Fetch Feed URL http://feeds.reuters.com/reuters/wo Truncate Truncate feed after 3 Truncate Truncate feed after 3 Truncate Truncate feed after 3 Truncate Truncate feed after 3 Union Pipe Output
65 clones
B (clean)
 Fetch Feed URL http://newsrss.bbc.co.uk/rss/news Fetch Feed URL http://news.google.com/?output=a Fetch Feed URL http://rss.news.yahoo.com/rss/tops Fetch Feed URL http://feeds.reuters.com/reuters/wo String Builder String 3 Fetch Feed URL http://feeds.feedburner.com/Wikine Truncate Truncate feed after num Truncate Truncate feed after num Truncate Truncate feed after num Truncate Truncate feed after num Truncate Truncate feed after num Union Pipe Output
32 clones

Context: *Truncate modules in Pipe A have hard-coded field values, but receive values via wire in Pipe B*

Characteristic: *to update in the future (e.g., allow only 1 item per website, not 3)*

HIT 3. Preference Consecutive Order-Independent Operators

Table A.4: Consecutive Order-Independent Operators Smell and Merge Consecutive Operators Refactoring

A (clean)	**B (smelly)**
	
41 clones	41 clones

Fetch Feed
URL
http://www.jeni-online.com/index.p
eeds.feedburner.com/AxisIncNews
Regex
Use regular expression patterns here:
Rules
In item.title replace ^(.+) with JENI Latest - ${title g s m
In item.description replace ^(.+)(\n|.)*- with g s m
Sort
Sort by
item.pubDate in descending order
Date Builder
Date: 20 days ago
Filter
Block items that match all of the following
Rules
item.pubDate is before text
Pipe Output

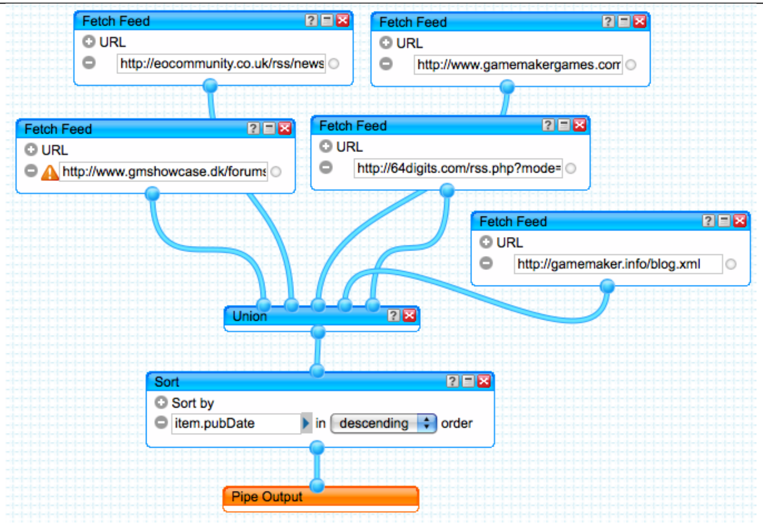
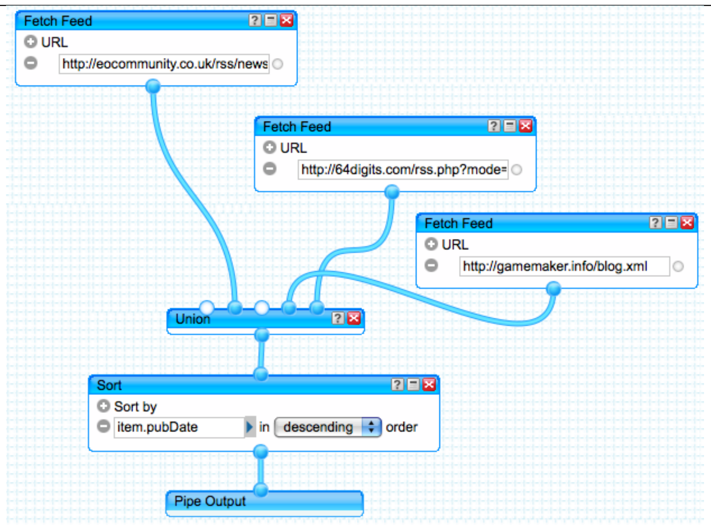
Fetch Feed
URL
http://www.jeni-online.com/index.p
eeds.feedburner.com/AxisIncNews
Regex
Use regular expression patterns here:
Rules
In item.title replace ^(.+) with JENI Latest - ${title g s m
Regex
Use regular expression patterns here:
Rules
In item.description replace ^(.+)(\n|.)*- with text g s m
Sort
Sort by
item.pubDate in descending order
Date Builder
Date: 20 days ago
Filter
Block items that match all of the following
Rules
item.pubDate is before text
Pipe Output

Context: *Rules in Regex modules modify a specified field's content (e.g., item.title), replacing instances of a pattern* $(\hat{(}.+))$ *with some text (JENI Latest -).*

Characteristic: *to understand*

HIT 4. Preference Invalid Source

Table A.5: Invalid Source Smell and Remove Deprecated Sources Refactoring

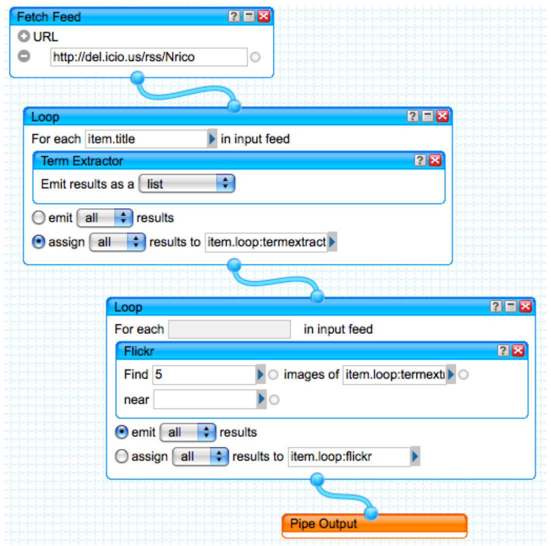
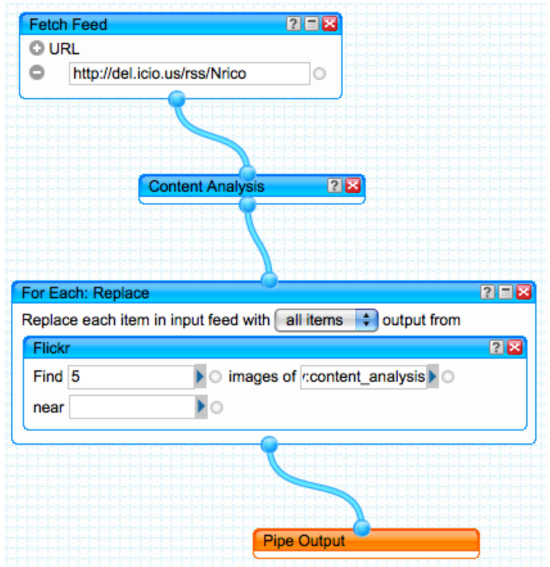
A (smelly)
 Fetch Feed URL http://eocommunity.co.uk/rss/news Fetch Feed URL http://www.gamemakergames.com Fetch Feed URL http://www.gmshowcase.dk/forums Fetch Feed URL http://64digits.com/rss.php?mode= Fetch Feed URL http://gamemaker.info/blog.xml Union Sort — Sort by item.pubDate in descending order Pipe Output
12 clones
B (clean)
 Fetch Feed URL http://eocommunity.co.uk/rss/news Fetch Feed URL http://64digits.com/rss.php?mode= Fetch Feed URL http://gamemaker.info/blog.xml Union Sort — Sort by item.pubDate in descending order Pipe Output
12 clones

Context: *Websites can be deleted, causing 404 errors, like these 2 in Pipe A: http://www.gamemakergames.com and http://www.gmshowcase.dk/forums.*

Characteristic: *to update in the future*

HIT 5. Preference Deprecated Modules

Table A.6: Deprecated Module Smell and Replace Deprecated Modules Refactoring

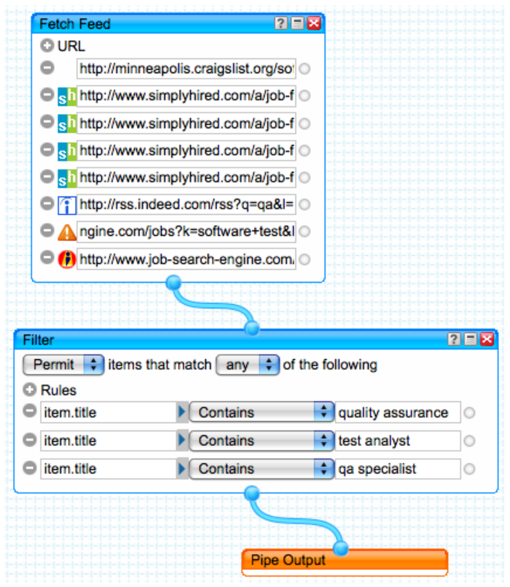
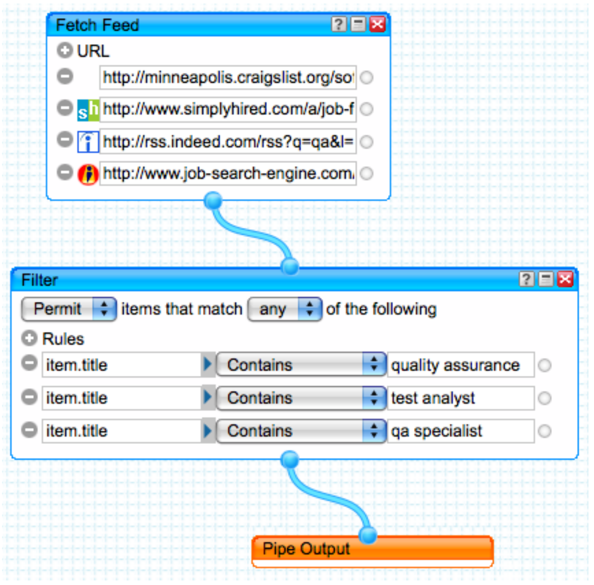
A (clean)	B (smelly)
 Fetch Feed: URL http://del.icio.us/rss/Nrico Loop: For each item.title in input feed Term Extractor: Emit results as a list emit all results [✓ assign] all results to item.loop:termextract Loop: For each in input feed Flickr: Find 5 images of item.loop:termexti near [✓ emit] all results assign all results to item.loop:flickr Pipe Output	 Fetch Feed: URL http://del.icio.us/rss/Nrico Content Analysis For Each: Replace: Replace each item in input feed with all items output from Flickr: Find 5 images of r:content_analysis near Pipe Output
128 clones	65 clones

Context: *Components are sometimes deprecated and replaced with improved features. In Pipe B, Content Analysis and For Each: Replace were deprecated.*

Characteristic: *for others to understand*

HIT 6. Preference Noisy Module

Table A.7: Duplicate Field Smell and Clean Up Module Refactoring

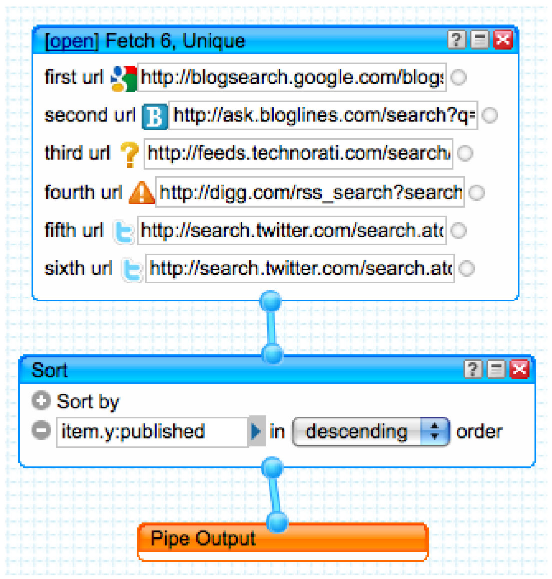
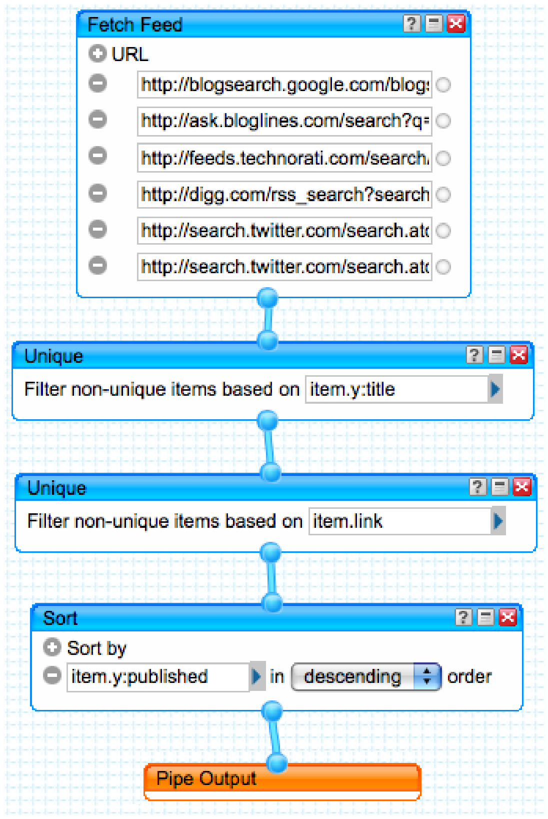
A (smelly)	B (clean)
 Fetch Feed URL http://minneapolis.craigslist.org/so http://www.simplyhired.com/a/job-f http://www.simplyhired.com/a/job-f http://www.simplyhired.com/a/job-f http://www.simplyhired.com/a/job-f http://rss.indeed.com/rss?q=qa&l= ngine.com/jobs?k=software+test&l http://www.job-search-engine.com. Filter Permit items that match any of the following Rules item.title Contains quality assurance item.title Contains test analyst item.title Contains qa specialist Pipe Output	 Fetch Feed URL http://minneapolis.craigslist.org/so http://www.simplyhired.com/a/job-f http://rss.indeed.com/rss?q=qa&l= http://www.job-search-engine.com. Filter Permit items that match any of the following Rules item.title Contains quality assurance item.title Contains test analyst item.title Contains qa specialist Pipe Output
7 clones	7 clones

Context: *Specifying the same website multiple times can lead to duplicate items in a pipe's output.*

Characteristic: *to update in the future*

HIT 7. Preference Global Isomorphic Paths

Table A.8: Global Isomorphic Paths Smell and Extract Global Subpipe Refactoring

A (clean)	B (smelly)
	
74 clones	74 clones

A (clean):

[open] Fetch 6, Unique
first url http://blogsearch.google.com/blogs
second url http://ask.bloglines.com/search?q=
third url http://feeds.technorati.com/search
fourth url http://digg.com/rss_search?search
fifth url http://search.twitter.com/search.at
sixth url http://search.twitter.com/search.at

Sort
Sort by
item.y:published in descending order

Pipe Output

B (smelly):

Fetch Feed
URL
http://blogsearch.google.com/blogs
http://ask.bloglines.com/search?q=
http://feeds.technorati.com/search
http://digg.com/rss_search?search
http://search.twitter.com/search.at
http://search.twitter.com/search.at

Unique
Filter non-unique items based on item.y:title

Unique
Filter non-unique items based on item.link

Sort
Sort by
item.y:published in descending order

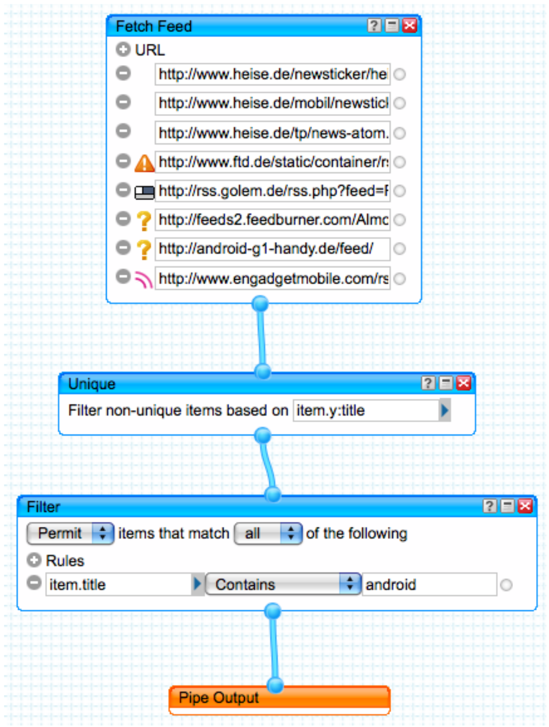
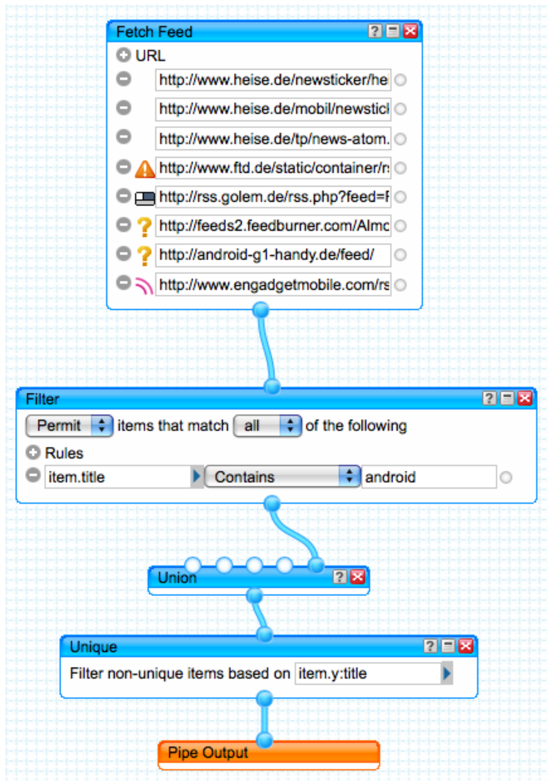
Pipe Output

Context: *In Pipe A, the "Fetch 6, Unique" subpipe module gathers the content of the six specified URLs and removes items that have duplicate titles or links.*

Characteristic: *to understand*

HIT 8. *Preference Unnecessary Module and Non-conforming Module Orderings*

Table A.9: Unnecessary Module and Non-conforming Module Orderings Smells and Normalize Order of Operations Refactoring

A (clean)	**B (smelly)**
	
13 clones	13 clones

A (clean):

Fetch Feed
URL
http://www.heise.de/newsticker/he
http://www.heise.de/mobil/newsticl
http://www.heise.de/tp/news-atom.
http://www.ftd.de/static/container/r:
http://rss.golem.de/rss.php?feed=f
http://feeds2.feedburner.com/Almc
http://android-g1-handy.de/feed/
http://www.engadgetmobile.com/rs

Unique
Filter non-unique items based on item.y:title

Filter
Permit items that match all of the following
Rules
item.title Contains android

Pipe Output

B (smelly):

Fetch Feed
URL
http://www.heise.de/newsticker/he
http://www.heise.de/mobil/newsticl
http://www.heise.de/tp/news-atom.
http://www.ftd.de/static/container/r:
http://rss.golem.de/rss.php?feed=f
http://feeds2.feedburner.com/Almc
http://android-g1-handy.de/feed/
http://www.engadgetmobile.com/rs

Filter
Permit items that match all of the following
Rules
item.title Contains android

Union

Unique
Filter non-unique items based on item.y:title

Pipe Output

Context: *The majority of the most popular pipes in the Yahoo! Pipes repository place the Unique module before the Filter module.*

Characteristic: *for others to understand*

A.2.2 Output Analysis

A HIT in this category has a structure like the one shown here. There are two HITs divided into two pairs. Section 9 shows two pipes with the same output, one refactored and one smelly. The same is true with Section 10. Each pipe is contained in its own HIT, but the options of output are the same. The points of variability are the pipe image the context (i.e., Pipes with different structures can generate the same output) and answers to the first question about the pipe's output.

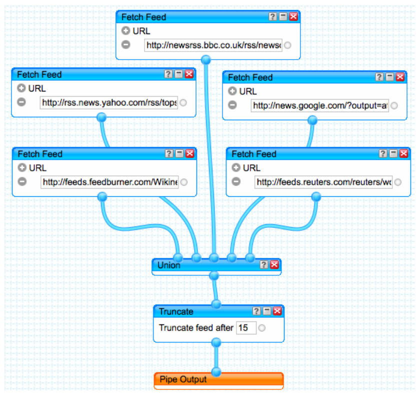

Fetch Feed
URL
http://newsrss.bbc.co.uk/rss/newsc
Fetch Feed
URL
http://rss.news.yahoo.com/rss/tops
Fetch Feed
URL
http://news.google.com/?output=a
Fetch Feed
URL
http://feeds.feedburner.com/Wikin
Fetch Feed
URL
http://feeds.reuters.com/reuters/wo
Union
Truncate
Truncate feed after 15
Pipe Output

Figure A.2: Output Analysis Example Pipe

Click the image to open a larger view.

Take some time to understand the behavior of each pipe. To answer questions 1 and 2 below, consider the following context:

Pipes with different structures can generate the same output.

1. Select the answer that most closely resembles the pipe's output.
   a) Option 1
   b) Option 2
   c) . . .
2. Justify your answer (you must use at least 10 words in your explanation):

HIT 9. Evaluating Isomorphic Paths

HIT 9.1. Clean Pipe

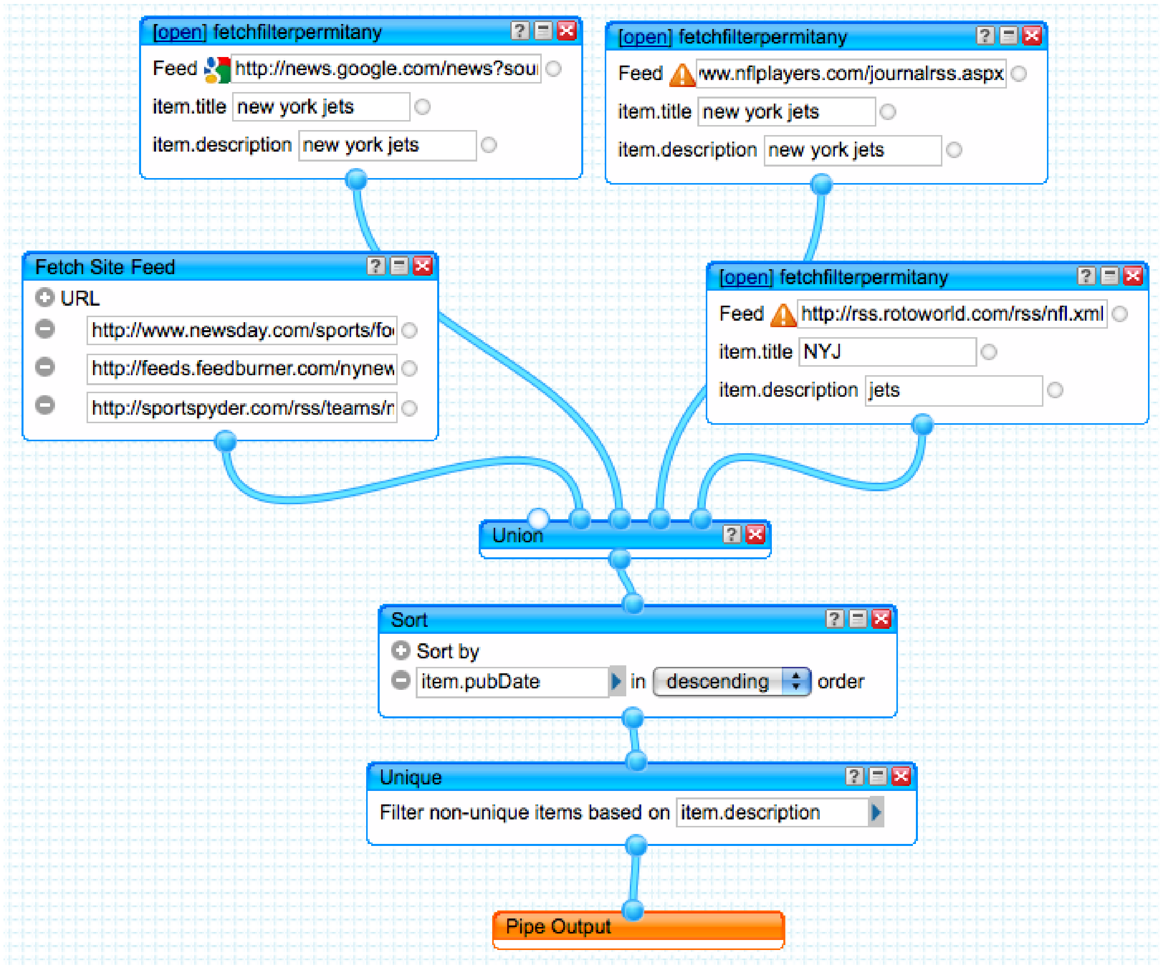
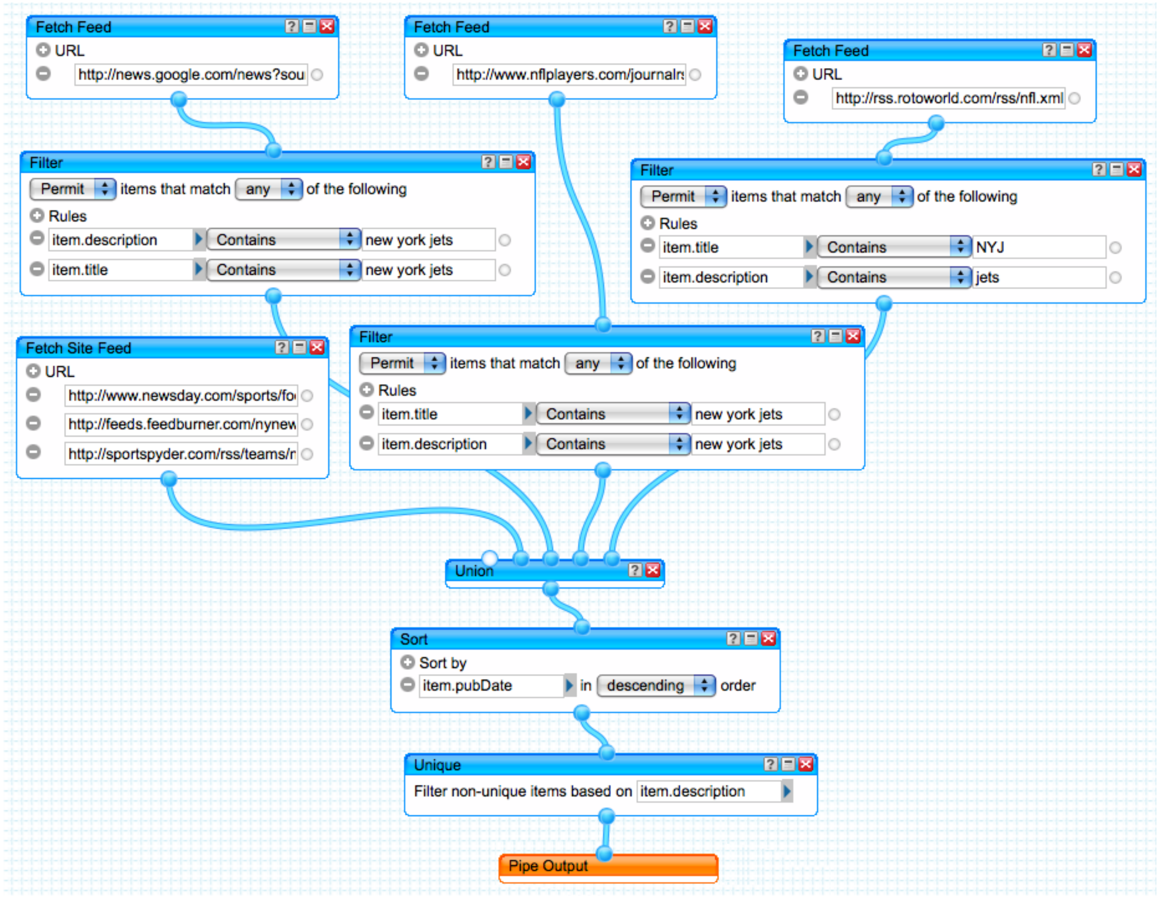

[open] fetchfilterpermitany
Feed http://news.google.com/news?sou
item.title new york jets
item.description new york jets
[open] fetchfilterpermitany
Feed ww.nflplayers.com/journalrss.aspx
item.title new york jets
item.description new york jets
Fetch Site Feed
URL
http://www.newsday.com/sports/fo
http://feeds.feedburner.com/nynew
http://sportspyder.com/rss/teams/r
[open] fetchfilterpermitany
Feed http://rss.rotoworld.com/rss/nfl.xml
item.title NYJ
item.description jets
Union
Sort
Sort by
item.pubDate in descending order
Unique
Filter non-unique items based on item.description
Pipe Output

Figure A.3: Isomorphic Paths Smell and Extract Local Subpipe, Refactored

Context: Subpipe ”fetchfilterpermitany” behaves as follows: it gathers the content of the website specified in the Feed field and filters items based on the title or description.

HIT 9.2. Dirty Pipe


Fetch Feed
URL
http://news.google.com/news?sou

Fetch Feed
URL
http://www.nflplayers.com/journalr

Fetch Feed
URL
http://rss.rotoworld.com/rss/nfl.xml

Filter
Permit items that match any of the following
Rules
item.description Contains new york jets
item.title Contains new york jets

Filter
Permit items that match any of the following
Rules
item.title Contains NYJ
item.description Contains jets

Fetch Site Feed
URL
http://www.newsday.com/sports/fo
http://feeds.feedburner.com/nynew
http://sportspyder.com/rss/teams/r

Filter
Permit items that match any of the following
Rules
item.title Contains new york jets
item.description Contains new york jets

Union

Sort
Sort by
item.pubDate in descending order

Unique
Filter non-unique items based on item.description

Pipe Output

Figure A.4: Isomorphic Paths Smell and Extract Local Subpipe, Smelly

Context: A filter module can be configured to permit or block items with certain characteristics. When multiple rules are provided, the filter module can consider any or all of the rules.

1. *Select the answer that most closely resembles the pipe's output.*

   a) *The content of three websites, sorted based on publication date in descending order, and where items with duplicate descriptions have been removed.*

   b) **The content of six websites (three of which are filtered based on the content in the title or description), sorted based on pubDate, where items with duplicate descriptions have been removed.**

c) *The content of six websites, sorted based on publication date in descending order, then filtered based on the title and description of each item, where items with duplicate descriptions have been removed.*

d) *There are four sets of output. One from each of the three websites that go to the fetchfilterpermitany module, and one for the three websites that are specified in the Fetch Site Feed module. The output from the three websites is also sorted and filtered for uniqueness.*

HIT 10. Joined Generators Smells and Evaluating Unnecessary Abstraction

HIT 10.1. Dirty Pipe

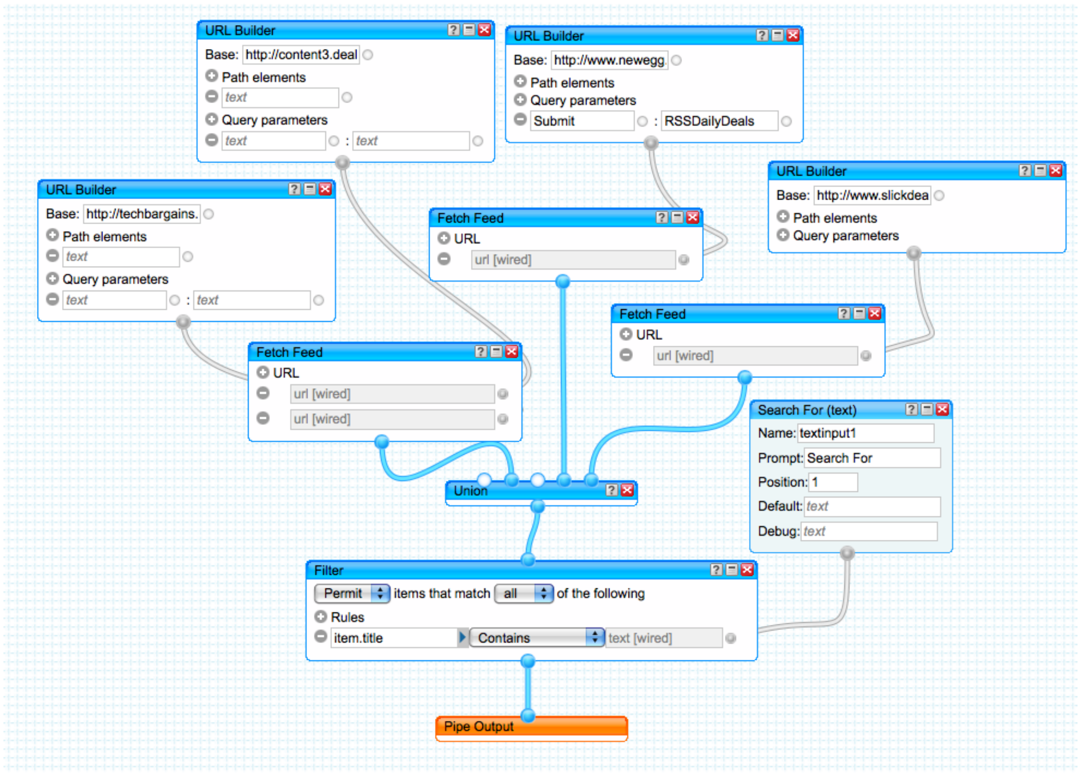
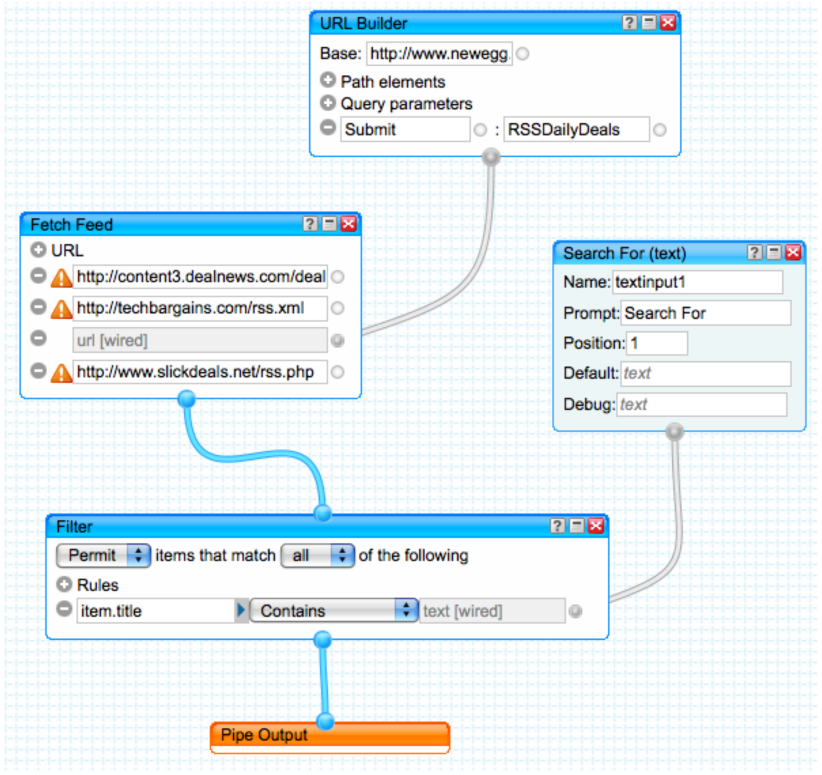


Figure A.5: Unnecessary Abstraction Smell and Joined Generators Smells and Push Down Module, Refactored

Context: A Search For module is a user-input module that gets a string from the user when the Pipe is run, using that string to set the value of fields connected via wire.

HIT 10.2. Clean Pipe



URL Builder
Base: http://www.newegg.
Path elements
Query parameters
Submit : RSSDailyDeals

Fetch Feed
URL
http://content3.dealnews.com/deal
http://techbargains.com/rss.xml
url [wired]
http://www.slickdeals.net/rss.php

Search For (text)
Name: textinput1
Prompt: Search For
Position: 1
Default: text
Debug: text

Filter
Permit items that match all of the following
Rules
item.title Contains text [wired]

Figure A.6: Unnecessary Abstraction and Joined Generators Smells and Push Down Module, Smelly

Context: A Search For module is a user-input module that gets a string from the user when the Pipe is run, using that string to set the value of fields connected via wire.

1. *Select the answer that most closely resembles the pipe's output.*

   a) *No Output*

   b) *All of the content of the websites specified in the URL Builders.*

   c) *The content of eight websites, filtered based on the presence of a user-defined value in the title of each item.*

   d) **The content of four websites, filtered based on the presence of a user-defined value in the title of each item.**

Appendix B

End User Study Additional Results

B.1 First Experiment: Preference

As a measure of understandability of the tasks themselves and the attention to detail given by each user group, we present the average time to completion for each associated HIT in Table B.1 for end users and degreed users, and in Table B.2 for users segmented based on qualification score.

Table B.1: Time to Completion of Preference HITs by Education (O_1)

HIT	**Type**	**End Users**	**Degreed Users**
1	Understanding	01:12	02:01
2	Maintainability	03:05	03:20
3	Understanding	07:43	02:01
4	Maintainability	05:34	01:57
5	Community Importance	02:45	03:38
6	Maintainability	02:16	01:50
7	Understanding	02:53	02:38
8	Community Importance	02:41	02:53

Table B.6 explores correlation coefficients across three other categories, organized per HIT.

Table B.2: Time to Completion of Preference HITs by Qualification Scores (O_2)

HIT	Type	High Score	Middle Score	Low Score
1	Understanding	02:11	01:15	01:28
2	Maintainability	03:36	02:25	03:24
3	Understanding	02:44	00:51	16:18
4	Maintainability	04:53	02:04	03:56
5	Community Importance	03:28	02:16	03:20
6	Maintainability	01:54	01:40	02:42
7	Understanding	02:57	03:04	02:11
8	Community Importance	02:56	01:57	03:19

Table B.3: Spearman's Correlation Coefficients for Preference Tasks

HIT	Qual Score & Time to Completion	Qual Score & Non-Smelly	Education & Non-Smelly
1	0.9968	0.7718	0.7296
2	0.9967	0.7797	0.6884
3	0.9971	0.4251	0.2308
4	0.9970	0.1889	0.2309
5	0.9980	0.0073	-0.0733
6	0.9953	0.6943	0.4958
7	0.9983	-0.4952	-0.6434
8	0.9977	0.8723	0.8510

B.2 Second Experiment: Correctness

Table B.4 shows the average time to completion for end users and degreed users on each of the output analysis HITs, and Table 3.14 gives the average times to completion for the participants grouped by qualification score.

We also explore the Spearman rank correlation coefficients among several variables for the output HITs, shown in Table B.6. The *Qual Score & Time to Completion* column indicates the correlation between a user's qualification score (scale of 0 to 8) and the time to completion for a particular HIT (in seconds). These variables are very strongly correlated. The *Qual Score & Correctness* column indicates the correlation between a user's qualifi-

Table B.4: Average Time to Completion of HITs by Education (O_1)

HIT	**Smell Type**	**Type**	**End Users**	**Degreed Users**
			Time to Completion	**Time to Completion**
9	Redund.	Clean 9.1	17:40	04:27
		Smelly 9.2	03:22	22:50
10	Redund. Lazy	Clean 10.2	09:04	04:14
		Smelly 10.1	27:56	14:55

Table B.5: Average Time to Completion of HITs by Qualification Score (O_2)

HIT	**Smell Type**	**Type**	**High Score**	**Middle Score**	**Low Score**
			Time to Completion	**Time to Completion**	**Time to Completion**
9	Redund.	Clean 9.1	04:09	12:08	05:50
		Smelly 9.2	22:46	00:00	03:26
10	Redund. Lazy	Clean 10.2	06:27	02:30	04:21
		Smelly 10.1	11:48	21:39	09:24

cation score and the correctness of their answer for a particular HIT. These correlations are also quite strong. The last column, *Education & Correctness*, indicates a correlation between the computer science education level of a user (scale of 0 to 4, where 4 is a degree) and the selection of the correct output.

Table B.6: Spearman's Correlation Coefficients for Output HITs

HIT	**Qual Score & Time to Completion**	**Qual Score & Correctness**	**Education & Correctness**
9.1 (clean)	0.9973	0.9571	0.8292
9.2 (dirty)	0.9979	0.9184	0.8661
10.2 (clean)	0.9973	0.9034	0.8759
10.1 (dirty)	0.9980	0.8965	0.8271

Appendix C

On the Semantic Correctness of the Refactorings

In this section, we show that each refactorings defined in Section 4.1 preserves a pipe's semantics. We say a refactoring is semantics preserving if the set of unique items that reaches the pipe's final $output$ module is the same before and after the refactoring is applied.

C.1 Overview and Approach

To ensure that the output of the pipe remains the same, we need to ensure that the items reaching the $output$ module are the same before and after the refactoring transformations, given the preconditions of the refactorings. Similar to the refactoring definitions, we let P_{before} represent a pipe that meets the refactoring precondition, and P_{after} represent a pipe that meets the refactoring postcondition. We also extend the notation to individual modules, using m_{before} to represent a module before a refactoring, and m_{after} to represent a module after a refactoring.

The refactorings involve several different transformations on the pipe, in which paths (p), modules (m), wires (w), and fields (f) are added and removed. For removal, we represent the transformation as follows: $P_{after} = P_{before} \setminus \{p, m, w, f\}$.[1] When adding artifacts to the pipe, it is represented as follows: $P_{after} = P_{before} \cup \{p, m, w, f\}$. We use the same notation when fields are removed from modules, $m_{after} = m_{before} \setminus f$, or when fields are added to modules, $m_{after} = m_{before} \cup f$.

To describe the output of a pipe, P, we use the notation $out(P)$ to represent the set of items returned by the execution of the pipe. We extend this notation to the output of modules, $out(m)$, paths, $out(p)$ and fields, $out(f)$, allowing us to compare between the output of different artifacts. For example, given the $output$ module $m \in P$, we can say that $out(P) = out(m)$ for every pipe P.

Definition 6. *A refactoring is semantics preserving if the output of the pipe before the refactoring,* $out(P_{before})$*, contains the* same set of unique items *as the output of the pipe after the refactoring,* $out(P_{after})$*, that is, if* $out(P_{before}) = out(P_{after})$*. The '$=$' operator indicates that the output of* P_{before} *and* P_{after} *are semantically equivalent, using this definition.*

Here, we give an example to illustrate the concept of semantic preservation given in Definition 6. If $out(P_{before}) = \{a, b, a, c\}$ and applying refactoring r results in $out(P_{after}) = \{a, b, c\}$, then r would be a semantics-preserving refactoring since $\{a, b, a, c\} = \{a, b, c\}$. However, if $out(P_{before}) = \{a, b, a, c\}$ and applying refactoring r' results in $out(P_{after}) = \{a, b, d\}$, then r' would not be semantics-preserving as $\{a, b, a, c\} \neq \{a, b, d\}$. We use a set to emphasize that it is not the order that matters, but rather the unique items that reach the output. The ordering of the items can always be modified by inserting a $sort$ module

[1]It is implied that $m \in P_{before}.\mathcal{M}$, $w \in P_{before}.\mathcal{W}$, and $f \in P_{before}.\mathcal{F}$, and $\forall mod \in p, mod \in P_{before}.\mathcal{M}$, but we chose a shortened syntax for brevity.

prior to the output, so in terms of the semantics, preservation of order is useful, but not necessary.

Some of the refactorings are trivially semantics preserving, as modifying a single module while preserving its behavior, or replacing a module with a semantically equivalent module are easily seen to preserve a pipe's behavior. That is, for any single module m that is altered by a refactoring, if $out(m_{before}) = out(m_{after})$, then it is clear that $out(P_{before}) = out(P_{after})$, and the refactoring preserved the pipe's semantics. And so, in refactorings that involve only a single module, it is sufficient to show that the output of the single module before and after the refactoring is preserved. With refactorings that involve multiple modules along a single path or the aggregation of multiple paths, it is sufficient to show that the output of the deepest module (i.e., the one closest to the $output$ module) is the same before and after the refactorings.

Section C.2 presents proof sketches for each of the refactorings defined in Section 4.1, illustrating how each transformation preserves the semantics of the pipes. For simplicity, we will consider a representative subset of the pipes language, specifically the generator module $fetch$, path-altering modules $union$ and $split$, operator modules $filter$ and $sort$, string setter module $strconcat$, and the $output$ module.

C.2 Proof Sketches

As previously defined in Section 3.1, a $Pipe$ is directed acyclic graph in which the modules are nodes and the wires are edges over which data flows. A module's output is either a single value, as when $m.type = setter.string$ or a list of items, as when $m.type = \{gen \mid pathAlt \mid op \mid output\}$. Here, we list some useful properties of the modules that will be used throughout this section.

- In a $fetch$ module m, $out(m) = \bigcup_{f \in m} out(f)$, where $out(f)$ is the set of items returned from querying f, an external data source.

- In a $union$ module m, we consider all the wires w such that $w.dest = m$, and define the set of all modules that are inputs to m, $inputMods(m) = \bigcup_{w \in \mathcal{W} | w.dest = m} w.src$, and define the output of m, $out(m) = \bigcup_{n \in inputMods(m)} out(n)$.

- In a $split$ module m, we consider all the wires such that $w.src = m$, and define the set of all output modules for m, $outputMods(m) = \bigcup_{w \in \mathcal{W} | w.src = m} w.dest$. We say the output of m, $out(m) \rightarrow \bigcup_{n \in outputMods(m)} in(n)$, where $\rightarrow$ indicates that $out(m)$ is copied and sent to each module in $outputMods(m)$. Since a copy of the input is sent along each output wire, $out(m) = in(m)$.

- In a $filter$ module m, $out(m) = in(m) \setminus \ell_m$, where ℓ_m is the list of items matching the conditions set by the fields in m. A $filter$ module generates ℓ_m by removing items that meet the criteria set by the fields in m. Here, we consider the behavior of m when it *removes* items that match *any* the criteria: $\ell_m = \bigcup_{f \in m} f.match = true$.

- In a $sort$ module m, $out(m) = in(m)$, since no items are added or removed.

- In a $strconcat$ module m, $out(m) = s$, where s is the concatenation of all fields in m. For example, if $| m.\mathcal{F} |= 3$, then $s = m.\mathcal{F}[1] + m.\mathcal{F}[2] + m.\mathcal{F}[3]$.

Refactoring 1: Clean Up Module

We have two cases of this refactoring: an empty field in a generator or setter, and a duplicate field in a generator. Each case modifies a single module in the pipe.

Smell 1.1: Empty Field

We consider an empty field in a $fetch$ module separately from an empty field in a $strconcat$ module. First, consider a $fetch$ module m_{before} such that $m_{before}.\mathcal{F} = \{f, g, h\}$ where f is an empty field. From the refactoring transformation, $m_{after} = m_{before} \setminus f$

$$\begin{array}{rcl|rcl}
 & P_{before} & & & P_{after} & \\
out(m) & = & out(f) \cup out(g) \cup out(h) & & & \\
out(f) & = & \{\} & & & \\
out(m) & = & out(g) \cup out(h) & out(m) & = & out(g) \cup out(h)
\end{array}$$

Additional non-empty fields will be treated similar to g and h.

Second, Consider a $strconcat$ module m_{before} with an empty field f. From the refactoring transformation, $m_{after} = m_{before} \setminus f$. Let us consider the case when $m.\mathcal{F}[i] = f =$“”, where $1 < i < n$, $n =\mid m.\mathcal{F} \mid$.

$$\begin{array}{rcl|rcl}
 & P_{before} & & & P_{after} & \\
out(m) & = & m.\mathcal{F}[1] + \cdots + m.\mathcal{F}[i-1] & & & \\
 & & +m.\mathcal{F}[i] & & & \\
 & & +m.\mathcal{F}[i+1] + \cdots + m.\mathcal{F}[n] & & & \\
m.\mathcal{F}[i] & = & \text{“”} & & & \\
out(m) & = & m.\mathcal{F}[1] + \cdots + m.\mathcal{F}[i-1] & out(m) & = & m.\mathcal{F}[1] + \cdots + m.\mathcal{F}[i-1] \\
 & & +m.\mathcal{F}[i+1] + \cdots + m.\mathcal{F}[n] & & & +m.\mathcal{F}[i+1] + \cdots + m.\mathcal{F}[n]
\end{array}$$

As shown for both the $fetch$ and $strconcat$ modules, $out(m_{before}) = out(m_{after})$ and $out(P_{before}) = out(P_{after})$.

Smell 1.2: Duplicate Field

Consider a $fetch$ module m_{before} such that $m_{before}.\mathcal{F} = \{f, g, h\}$ where $f.value = g.value$. From the refactoring, $m_{after} = m_{before} \setminus f$.

$$\begin{array}{rcl|rcl} & P_{before} & & & P_{after} & \\ out(m) & = & out(f) \cup out(g) \cup out(h) & & & \\ out(f) & = & out(g) & & & \\ out(m) & = & out(g) \cup out(g) \cup out(h) & & & \\ out(m) & = & out(g) \cup out(h) & out(m) & = & out(g) \cup out(h) \end{array}$$

As is shown, $out(m_{before}) = out(m_{after})$, and so $out(P_{before}) = out(P_{after})$. If m contains other fields, then each additional field is treated just as h is in this proof sketch.

Refactoring 2: Remove Non-Contributing Module

The precondition of this refactoring is one of the cases in Smell 2. In each case, a single module is removed.

Case 2.1. Disconnected, Dangling, or Swaying

We consider three scenarios for this case, where each scenario comes from the preconditions for this refactoring. First, consider a disconnected module m. From the refactoring transformation, $P_{after} = P_{before} \setminus m$. Second, consider a dangling module m connected to P_{before} by wire w. From the refactoring, $P_{after} = P_{before} \setminus \{m, w\}$. Third, consider a swaying path altering or operator module m and wire w connecting m to P_{before}. From the refactoring transformation, $P_{after} = P_{before} \setminus \{m, w\}$. In all cases, since m does contribute to the output of the pipe, $out(P_{before}) = out(P_{after})$.

Case 2.2. Lazy Module

There are three separate preconditions that can trigger this refactoring. For ineffectual path altering module m in which there is exactly one wire $w \mid w.dest = m$, then $out(m) = out(w.src) = in(m)$. For an inoperative module m with outgoing wire w, such as a $filter$

module with no fields, then $out(m) = in(m)$. For unnecessary redirection in which a $strconcat$ module m with output wire w has only one field f that receives its value via wire, then again $out(m) = in(m)$. In all these cases, the refactoring transformation removes m and w, so $P_{after} = P_{before} \setminus \{m, w\}$ and $out(P_{before}) = out(P_{after})$.

Refactoring 3: Push Down Module

For a $strconcat$ module m in which none of the field values are wired, the value of $s = out(m)$ can be generated statically and used set the value for all wired fields that received their value from m. The refactoring transformation removes m and sets field values to s. From the refactoring, $P_{after} = P_{before} \setminus m$ and $\forall w_{before} \in P_{before}.\mathcal{W} \mid out\_wire(m, w_{before}), (w_{after}.fld).value = s$ & $w_{before} \notin P_{after}$. This leaves m disconnected in P_{after} so m can be trivially removed. It follows that $out(P_{before}) = out(P_{after})$.

Refactoring 4: Merge Redundant Modules

For this refactoring, modules that lie along the same path or that are on different paths connected with a $union$ module, are merged. We consider each case individually.

Case 4.1: Merge Consecutive Operators

Consider two $filter$ modules, m and n, and wire w, where $out\_wire(n, w)$ (mapping to m_i, m_j, and w_j in Figure 4.2, respectively). From the refactoring transformation, $m_{after}.\mathcal{F} = m_{before}.\mathcal{F} \cup n_{before}.\mathcal{F}$ and $P_{after} = P_{before} \setminus \{n, w\}$. Since n is being removed, the final condition is that $out(p_{before}) = out(m_{after})$, where path $p_{before} = [m_{before}, n_{before}]$. This is because m_{after} absorbs the fields from n_{before}, n_{before} is the immediate postdominator of m_{before}, and $n_{before} \notin P_{after}$.

$$\begin{array}{rcl|rcl}
\multicolumn{3}{c|}{P_{before}} & \multicolumn{3}{c}{P_{after}} \\
out(m) & = & in(m) \setminus \ell_m & & & \\
out(n) & = & in(n) \setminus \ell_n & & & \\
in(n) & = & out(m) & & & \\
out(n) & = & out(m) \setminus \ell_n & & & \\
out(n) & = & (in(m) \setminus \ell_m) \setminus \ell_n & & & \\
out(n) & = & in(m) \setminus (\ell_m \cup \ell_n) & out(m) & = & in(m) \setminus (\ell_m \cup \ell_n)
\end{array}$$

As shown, the items reaching the output of n_{before} are the same as those reaching m_{after}, which means $out(p_{before}) = out(m_{before})$, and so $out(P_{before}) = out(P_{after})$.

Case 4.2: Merge Path Altering Modules

Consider $union$ modules m and o connected by wire w (mapping to m_i, m_j and w_i in Figure 4.3, respectively), where $inputMods(m) = \{j, k\}$ and $inputMods(o) = \{m, l, n\}$. From the refactoring transformation, $P_{after} = P_{before} \setminus \{m, w\}$.

$$\begin{array}{rcl|rcl}
\multicolumn{3}{c|}{P_{before}} & \multicolumn{3}{c}{P_{after}} \\
out(m) & = & out(j) \cup out(k) & & & \\
out(o) & = & out(m) \cup out(l) \cup out(n) & & & \\
out(o) & = & (out(j) \cup out(k)) \cup out(l) & out(o) & = & out(j) \cup out(k) \cup out(l) \\
 & & \cup out(n) & & & \cup out(n)
\end{array}$$

As shown, $out(o_{before}) = out(o_{after})$, so consequently, $out(P_{before}) = out(P_{after})$. If m has additional input modules, then the output of each module would be added via union to $out(m_{before})$, and also $out(o_{after})$. If o has additional input modules, these are added via union to $out(o_{before})$ and $out(o_{after})$.

We also consider *split* modules m and o connected by wire w where $outputMods(m) = \{j, o\}$, $outputMods(o) = \{k, l\}$, and $inputMods(m) = n$. From the refactoring, $P_{after} = P_{before} \setminus \{m, w\}$.

$$\begin{array}{rcl|rcl}
 & P_{before} & & & P_{after} & \\
out(m) & \rightarrow & in(j) \cup in(o) & & & \\
out(o) & \rightarrow & in(k) \cup in(l) & & & \\
out(o) & = & in(o) & & & \\
out(m) & \rightarrow & in(j) \cup (in(k) \cup in(l)) & & & \\
out(o) & = & out(m) & & & \\
out(o) & \rightarrow & in(j) \cup (in(k) \cup in(l)) & out(o) & \rightarrow & in(j) \cup in(k) \cup in(l)
\end{array}$$

As shown, $out(o_{before}) = out(o_{after})$, so consequently, $out(P_{before}) = out(P_{after})$. Additional output modules for m will be treated similar to j, and additional output modules for o will be treated similar to k or l.

Case 4.3: Merge Subsequent Operators

Consider two identical operator modules, m and n, and wire w such that $out\_wire(m, w)$ (mapping to m_i, m_j, and w_j in Figure 4.4, respectively), in which n postdominates m. According to the precondition, m and n may be separated by $op.orderIndep$ or $union$ modules. We consider a path $p =$ [m,a,b,n] where $a.name = filter$ and $b.name = union$, and $inputMods(b) = \{a, c\}$. From the refactoring, $P_{after} = P_{before} \setminus \{m, w\}$, and so for our example, $in(m_{before}) = in(a_{after})$.

$$\begin{array}{rcl|rcl}
 & P_{before} & & & P_{after} & \\
out(m) & = & in(m) \setminus \ell_m & & & \\
out(a) & = & in(a) \setminus \ell_a & & & \\
in(a) & = & out(m) & & & \\
out(a) & = & (in(m) \setminus \ell_m) \setminus \ell_a & out(a) & = & in(a) \setminus \ell_a \\
out(b) & = & out(a) \cup out(c) & out(b) & = & out(a) \cup out(c) \\
out(b) & = & ((in(m) \setminus \ell_m) \setminus \ell_a) \cup out(c) & out(b) & = & (in(a) \setminus \ell_a) \cup out(c) \\
out(n) & = & in(n) \setminus \ell_n & out(n) & = & in(n) \setminus \ell_n \\
in(n) & = & out(b) & in(n) & = & out(b) \\
out(n) & = & (((in(m) \setminus \ell_m) \setminus \ell_a) \cup out(c)) \setminus \ell_n & out(n) & = & ((in(a) \setminus \ell_a) \cup out(c)) \setminus \ell_n \\
out(n) & = & (in(m) \setminus (\ell_m \cup \ell_a \cup \ell_n)) & & & \\
 & & \cup(out(c) \setminus \ell_n) & & & \\
\ell_m & = & \ell_n & & & \\
out(n) & = & (in(m) \setminus (\ell_a \cup \ell_n)) & out(n) & = & (in(a) \setminus (\ell_a \cup \ell_n)) \\
 & & \cup(out(c) \setminus \ell_n) & & & \cup(out(c) \setminus \ell_n)
\end{array}$$

As shown, since $in(m_{before}) = in(a_{after})$, $out(n_{before}) = out(n_{after})$ and $out(P_{before}) = out(P_{after})$. This proof sketch generalizes for any number and any order of $union$ and $operator$ modules along the path separating m and n. Additional $operator$ modules, such as q, would be treated similar to a, and the $out(n)$ would also have ℓ_q removed in P_{before} and P_{after}. Additional $union$ modules would be treated similar to b, and additional source modules into a union module would be treated like c.

Refactoring 5: Collapse Duplicate Paths

These refactorings involve collapsing two paths in a pipe into a single path.

Case 5.1: Joined Generators

Consider $fetch$ module m with fields a and b, $fetch$ module n with fields c and d, $union$ module u with $inputMods(u) = \{m, n, q\}$, with wire w connecting m to u (mapping to m_i, m_j, m_u, and w_i, respectively, as illustrated in Figure 4.5). From the refactoring, $n_{after}.\mathcal{F} = m_{before}.\mathcal{F} \cup n_{before}.\mathcal{F}$, and $P_{after} = P_{before} \setminus \{m, w\}$.

$$\begin{array}{rcl|rcl}
\multicolumn{3}{c|}{P_{before}} & \multicolumn{3}{c}{P_{after}} \\
out(m) & = & out(a) \cup out(b) & & & \\
out(n) & = & out(c) \cup out(d) & out(n) & = & out(a) \cup out(b) \\
 & & & & & \cup out(c) \cup out(d) \\
out(u) & = & out(m) \cup out(n) \cup out(q) & out(u) & = & out(n) \cup out(q) \\
out(u) & = & (out(a) \cup out(b)) & out(u) & = & (out(a) \cup out(b) \\
 & & \cup(out(c) \cup out(d)) & & & \cup out(c) \cup out(d)) \\
 & & \cup out(q) & & & \cup out(q)
\end{array}$$

As is shown, $out(u_{before}) = out(u_{after})$, and so $out(P_{before}) = out(P_{after})$. If u has additional input modules, the output of each additional input module would be added via union everywhere $out(q)$ appears in the proof. If u has fewer input modules, for example the absence of q, then $out(q)$ can be removed without impacting the final equality condition.

Case 5.2: Identical Parallel Operator Pairs

Consider $fetch$ module k with fields a and b, $fetch$ module l with field c, identical $sort$ modules m and n, $union$ module u where $inputMods(u) = \{m, n, o\}$, wire w connecting k to m, and wire y connecting m to u, mapping to m_k, m_l, m_i, m_j, m_u, w_k, and w_i, respectively, as illustrated in Figure 4.6. From the refactoring, $P_{after} = P_{before} \setminus \{k, m, w, y\}$, and $l_{after}.\mathcal{F} = k_{before}.\mathcal{F} \cup l_{before}.\mathcal{F}$

$$
\begin{array}{rcl|rcl}
\multicolumn{3}{c|}{P_{before}} & \multicolumn{3}{c}{P_{after}} \\
out(k) & = & out(a) \cup out(b) & & & \\
out(l) & = & out(c) & out(l) & = & out(a) \cup out(b) \cup out(c) \\
out(m) & = & in(m) \setminus \ell_m & & & \\
in(m) & = & out(k) & & & \\
out(m) & = & (out(a) \cup out(b)) \setminus \ell_m & & & \\
out(n) & = & in(n) \setminus \ell_n & out(n) & = & in(n) \setminus \ell_n \\
in(n) & = & out(l) & in(n) & = & out(l) \\
out(n) & = & (out(c)) \setminus \ell_n & out(n) & = & (out(a) \cup out(b) \cup out(c)) \setminus \ell_n \\
out(u) & = & out(m) \cup out(n) \cup out(o) & out(u) & = & out(n) \cup out(o) \\
out(u) & = & ((out(a) \cup out(b)) \setminus \ell_m) & & & \\
 & & \cup(out(c) \setminus \ell_n) \cup out(o) & & & \\
\ell_m & = & \ell_n & & & \\
out(u) & = & (out(a) \cup out(b) \cup out(c)) \setminus \ell_n & out(u) & = & (out(a) \cup out(b) \cup out(c)) \setminus \ell_n \\
 & & \cup out(o) & & & \cup out(o)
\end{array}
$$

As is shown, $out(u_{before}) = out(u_{after})$, and so $out(P_{before}) = out(P_{after})$. In this example, we consider merging paths with two modules connected by a union, but it could be generalized to longer paths. For each additional operator module along the paths, they would be treated similarly to m and n, where the module along the path including m would be removed (as m is removed). Additional input modules for u would be treated just like o, and o could be removed from the proof sketch without altering its validity.

Refactoring 6: Pull Up Module

This refactoring works in the opposite direction as *Refactoring 3: Push Down Module*. Here, the strings from fields in multiple modules are abstracted into a separate module so the fields can receive their values via wire. Consider the case of two modules, n and o, with duplicated fields f and h. From the refactoring, $P_{after} = P_{before} \cup \{m, g\}$, where $g.value = f.value$. Additionally, $\forall f \in P_{before}.\mathcal{F} \mid f.value = g.value, \exists w \in P_{after}.\mathcal{W} \mid field\_wire(owner(f), w)$. Through the refactoring, no fields or modules are removed from the pipe. For fields f and h, $owner(f_{before}) = owner(f_{after})$ and $owner(h_{before}) = owner(h_{after})$. For the modules n and o, $out(n_{before}) = out(n_{after})$, and $out(o_{before}) = out(o_{after})$. In P_{after}, fields f and h receive their values via wire from a new module m, where $out(m) = f_{before}.value = h_{before}.value$. This preserves the output of the entire pipe, so $out(P_{after}) = out(P_{before})$. If there are more duplicated field values in the pipe, then a wire is added from m to each duplicated field.

Refactoring 7, 11: Extract Local Subpipe, Extract Global Subpipe

For this refactoring, we need to show that for some path p, we can create a separate pipe, P_{new}, that encodes the behavior as p, and can replace p while preserving a pipe's semantics. That is, we must show that $out(p) = out(P_{new})$.

In the refactoring, the pipe P_{new} is generated by first copying p into a new pipe. The next step is to add an $output$ module, o, and a wire such that $p(last)$ and o are now connected. We now have a complete pipe, P_{new}, that is a replica of p, so $out(p) = out(P_{new})$. The next step is to parameterize all the fields in P_{new} so that when $subpipe(P_{new})$ is added to a pipe, it can be populated with appropriate values that will encode the behavior of the path being replaced. This is done by adding a $user.setter$ module for every field in P_{new} that can receive a value via wire. For each $user.setter$ module in P_{new}, $subpipe(P_{new})$ contains a wireable field. These fields in $subpipe(P_{new})$ are populated in P_{after} using the values of the wireable fields from p. In this way, for every field $f \in P_{new} \mid f.wireable = false$, this value is set by the copy of p in P_{new}. For every field $f \in P_{new} \mid f.wireable = true$, it can be set in P_{after}, so that $out(p) = out(subpipe(P_{new}))$.

If we create P_{new} for some path $p \in P_{before}$, then all paths isomorphic to p can be similarly replaced. For every path p replaced through this refactoring, we see that $P_{after} = P_{before} \setminus p$, and $P_{after} = P_{before} \cup subpipe(P_{new})$. Since $out(p) = out(subpipe(P_{new}))$, then $out(P_{before}) = out(P_{after})$.

Refactoring 8: Replace Deprecated Modules

Consider a deprecated module, m_{dep} that can be replaced by a module or path, M_{new} using a function $replace : m \rightarrow M$ that takes a deprecated module and replaces it with a semantically equivalent module or path. From the refactoring, $P_{after} = P_{before} \setminus m_{dep}$ and $P_{after} = P_{before} \cup M_{new}$. And so, as long as $out(m_{dep}) = out(M_{new})$, $out(P_{before}) = out(P_{after})$. The correctness of this refactoring is dependent on the correctness of the $replace$ function, which is based on Yahoo!'s documentation.

Refactoring 9: Remove Deprecated Sources

By the definition of a deprecated source, it is non-contributing to the module. If field f refers to a deprecated external source, then $out(f) = \{\}$. In the refactoring, we see that $P_{after} = P_{before} \setminus f$. The only impacted module in this refactoring is $m = owner(f)$. Through the refactoring, $m_{after} = m_{before} \setminus f$. Since $out(f) = \{\}$, then $out(m_{before}) = out(m_{after})$, and $out(P_{before}) = out(P_{after})$.

Refactoring 10: Normalize Order of Operations

Consider a non-conforming path p and a path $ppres$ that will replace p. From the refactoring, $P_{after} = P_{before} \setminus p$ and $P_{after} = P_{before} \cup ppres$, where $bag(p) = bag(ppres)$. Since all the modules are the same between p and $ppres$, and only the ordering of modules in p

has changed, we just need to show that $out(p) = out(ppres)$. Consider an example where n is a filter module, o is a sort module, $p = [n, o]$, and $ppres = [o, n]$.

$$\begin{array}{rcl|rcl}
\multicolumn{3}{c}{P_{before}} & \multicolumn{3}{c}{P_{after}} \\
in(n) & = & in(p) & in(o) & = & in(ppres) \\
out(n) & = & in(n) \setminus \ell_n & out(o) & = & in(o) \\
out(n) & = & in(p) \setminus \ell_n & out(o) & = & in(ppres) \\
out(o) & = & in(o) & out(n) & = & in(n) \setminus \ell_n \\
in(o) & = & out(n) & in(n) & = & out(o) \\
out(o) & = & in(p) \setminus \ell_n & out(n) & = & in(ppres) \setminus \ell_n \\
out(p) & = & out(o) & out(ppres) & = & out(n) \\
out(p) & = & in(p) \setminus \ell_n & out(ppres) & = & in(ppres) \setminus \ell_n
\end{array}$$

Since $ppres \in P_{after}$ replaces $p \in P_{before}$, $in(ppres) = in(p)$. As shown, $out(p) = out(ppres)$, and so $out(P_{before}) = out(P_{after})$. If p contains additional filter modules (like n in the example), then $out(p) = in(p) \setminus (\ell_n \cup \ell_{filter})$, where ℓ_{filter} is the union of the set of items removed by each additional $filter$ module in $bag(p)$. Since $bag(p) = bag(ppres)$, ℓ_{filter} will have the same impact on $out(ppres)$, so the proof sketch holds. Additional sort modules can be trivially shown to preserve the semantics of the pipe, since $out(sort) = in(sort)$ for any $sort$ module.

Bibliography

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

[2] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–279, 2005.

[3] Luciano Baresi and Reiko Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. *Lecture Notes in Computer Science*, pages 402–429, 2002.

[4] Don S. Batory. Program Refactoring, Program Synthesis, and Model-Driven Development. In *Compiler Construction: 16th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software*, pages 156–171, 2007.

[5] Math - The Commons Math User Guide - Statistics. http://commons.apache.org/math/userguide/stat.html, May 2010.

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

[7] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proceedings of the 31st International Conference on Software Engineering*, pages 397–407, 2009.

[8] Eclipse.org. http://eclipse.org/, May 2010.

[9] Feed Rinse. http://feedrinse.com/, January 2010.

[10] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] Lars Grammel and Margaret-Anne Storey. An End User Perspective on Mashup Makers. Technical Report DCS-324-IR, University of Victoria, September 2008.

[12] Warren Harrison. From the Editor: The Dangers of End-User Programming. *IEEE Software*, pages 5–7, 2004.

[13] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering*, pages 274–283. ACM, 2005.

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

[15] M. Cameron Jones and Elizabeth F. Churchill. Conversations in Developer Communities: A Preliminary Analysis of the Yahoo! Pipes Community. In *Proceedings of the Fourth International Conference on Communities and Technologies*, pages 195–204, 2009.

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

[17] JSON in Java. http://www.json.org/java/index.html, July 2009.

[18] Hannes Kegel and Friedrich Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the 30th International Conference on Software Engineering*, pages 431–440, 2008.

[19] Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for Parameterizing Java Classes. In *Proceedings of the 29th International Conference on Software Engineering*, pages 437–446, 2007.

[20] Kivati. http://kivati.com/, January 2010.

[21] Ko, Andrew J. and Myers, Brad A. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering*, pages 301–310, 2008.

[22] Andhy Koesnandar, Sebastian Elbaum, Gregg Rothermel, Lorin Hochstein, Christopher Scaffidi, and Kathryn T. Stolee. Using Assertions to Help End-User Programmers Create Dependable Web Macros. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 124–134, 2008.

[23] Christian Köhler, Holger Lewin, and Gabriele Taentzer. Ensuring Containment Constraints in Graph-based Model Transformation Approaches. In *International Workshop on Graph Transformations and Visual Modeling Techniques*, pages 45–56, 2007.

[24] Jia Liu, Don S. Batory, and Christian Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proceedings of the 28th International Conference on Software Engineering*, pages 112–121, 2006.

[25] Amazon Mechanical Turk Command Line Tool Reference . http://docs.amazonwebservices.com/AWSMturkCLT/2008-08-02/, January 2010.

[26] Tom Mens, Niels Van Eetvelde, Dirk Janssens, and Serge Demeyer. Formalizing Refactorings with Graph Transformations. *Journal of Software Maintenance and Evolution*, 17(4):247–276, 2005.

[27] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[28] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

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

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

[31] C. Scaffidi, C. Bogart, M. Burnett, A. Cypher, B. Myers, and M. Shaw. Predicting reuse of end-user web macro scripts. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 93–100, 2009.

[32] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Symposium on Visual Languages and Human Centric Computing*, pages 207–214, 2005.

[33] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. *Lecture Notes in Computer Science*, pages 134–148, 2001.

[34] Gabriele Taentzer, Dirk Müller, and Tom Mens. Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation. In *Applications of Graph Transformations with Industrial Relevance*, pages 104–119, 2007.

[35] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for Reentrancy. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference*

and the ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 173–182, 2009.

[36] Jeffrey Wong and Jason Hong. What Do We ”Mashup” When We Make Mashups? In *Proceedings of the 4th International Workshop on End-User Software Engineering*, pages 35–39, 2008.

[37] xFruits. http://www.xfruits.com/, August 2009.