

Acknowledgements

My PhD has been a pleasant and fruitful journey which brought many life-lasting knowledge, experiences, lessons, memories, friendships, and of course, a pile of code and documents including this dissertation. As I am writing this, I am miles away from those who made this journey possible. So I am taking this opportunity to profusely thank them in writing.

Foremost, I would like to express my greatest gratitude and appreciation to Prof. Cesare Pautasso, from whom I learned many things. Being always available, he patiently guided me through ups and downs of my PhD, and believed in me when I was lost. It was a pleasure and a privilege to do my PhD under your supervision.

I am sincerely grateful to Daniele Bonetta, Marcin Nowak, Dr. Achille Peternier, Masiar Babazadeh, and Vasileios Triglianios for being friends, giving continuous constructive feedback on my work, proofreading my papers, and participating in my user studies. I wish you luck in your lives and hope to see you again.

I would like to thank Mauro Prevostini for giving me the opportunity to conduct a usability test on high school students, Dr. Monica Landoni for her valuable input into the design of the formative evaluation of my work, and Prof. Antonella De Angeli for hosting me in University of Trento, Italy, to properly conduct a user study of my system.

I am thankful to my best friends, Ehsan, Morteza, Lea, Marco, Andrea, and Silvia, who made the beautiful and wonderful city of Lugano socially pleasant for me. Your memory lingers on.

I profusely thank my parents, Akram and Hassan, and my brothers, Farid and Amir, without whom I wouldn't be where I am today. All I ever need is you by my side, so live forever.

Last but not least, I wish to thank my love of life, Negin, for her loving care and support. I am so lucky to have you in my life.

Contents

Contents	ix
List of Figures	xiii
List of Tables	xvii
1 Introduction	1
1.1 The Rise of “Citizen Developers”	2
1.2 The Potential of Web Mashups	3
1.3 Problem Statement	4
1.4 Thesis Statement	5
1.5 Contributions	6
1.6 Structure	7
2 A Survey of the State-of-the-art Mashup Tools	9
2.1 Defining Mashup Tools	9
2.2 Survey Methodology	11
2.3 Design Issues and Alternatives	11
2.3.1 Design Issue: Automation Degree	11
2.3.2 Design Issue: Liveness	13
2.3.3 Design Issue: Live Layout	15
2.3.4 Design Issue: Online Community	17
2.3.5 Design Issue: End-User Programming Technique	17
2.3.6 Design Issue: Visual Language	20
2.3.7 Design Issue: Design Distance	21
2.4 Survey Summary	24
2.5 The Need for Another Mashup Tool	24
3 NaturalMash: A Live Natural Mashup Tool	31
3.1 Design Decisions	31

3.2	NaturalMash Controlled Natural Language	34
3.2.1	Grammar and Syntax	35
3.2.2	Semantics	36
3.3	NaturalMash Component Meta-Model	37
3.3.1	Modeling Ingredient Technologies	38
3.3.2	Mapper and Rule	43
3.3.3	JSON Schema for The Component Model	44
3.3.4	Formal Description of ingredients in Natural Language	46
3.4	Ontology-based Data Integration Framework	47
3.4.1	Schema Specification and User-defined Data Formats	48
3.4.2	Integration Process	50
3.5	NaturalMash Composition Environment	52
3.5.1	Usage Scenario	55
4	The Architecture of NaturalMash	57
4.1	Design Challenges for Live Mashup Tools	57
4.2	A liveness-friendly Architecture	58
4.2.1	User Interface	59
4.2.2	Incremental Change Detector of Mashup Models	59
4.2.3	Compilation and Deployment	60
4.2.4	Client-server Communication	65
4.2.5	Mashup Runtime	66
5	Evaluating NaturalMash In-The-Lab	67
5.1	First Iteration	67
5.2	Second Iteration	69
5.2.1	Participants	69
5.2.2	Tasks and Methods	70
5.2.3	Results	71
5.2.4	Lessons Learned	71
5.3	Third Iteration	72
5.3.1	Participants	72
5.3.2	Tasks and Methods	73
5.3.3	Results	73
6	Summative Evaluation of NaturalMash	77
6.1	Evaluation Questions and Hypotheses	77
6.2	Evaluation Methods Overview	78
6.3	Data Collection Methods	78

6.3.1	Task-based Metrics	78
6.3.2	End-of-session Metrics	81
6.4	Technology	81
6.4.1	Participant Recruitment and Sample Collection	81
6.5	Evaluation Procedure	84
6.5.1	Starter Questions	84
6.5.2	Tasks	84
6.5.3	Post-Task Questions	86
6.5.4	Post-Session Questions	87
6.5.5	Wrap-up	87
6.6	Results	87
6.6.1	Task Performance Data	87
6.6.2	Self-reported Data	91
6.6.3	Logs	93
6.7	Conclusion and Lessons learned	96
7	Comparative Evaluation of Mashup Tools	99
7.1	A Benchmarking Framework for Mashup Tools	100
7.1.1	Benchmarking Strategy	102
7.2	Benchmarking the State-of-the-art Mashup Tools	102
7.2.1	Methods	105
7.2.2	Results and Discussion	107
8	Conclusion	109
8.1	Limitations	111
8.1.1	Comparative Usability Evaluation	111
8.1.2	Summative Usability Evaluation	112
8.1.3	Comparative Expressive Power Evaluation	112
8.2	Future Directions	113
A	Usability Evaluation Forms	115
A.1	Background Assessment Questionnaire	115
A.2	Exit Questionnaire: First Iteration Formative Evaluation	118
A.3	Exit Questionnaire: Second Iteration Formative Evaluation	120
A.4	Exit Questionnaire: Summative Evaluation	122
A.5	Post-task Class 1 Questionnaire: Summative Evaluation	123
A.6	Post-task Class 2 Questionnaire: Summative Evaluation	124
A.7	Post-task Class 3 Questionnaire: Summative Evaluation	125
A.8	Post-task Class 4 Questionnaire: Summative Evaluation	126

Bibliography**127**

Figures

1.1	The long tail of software applications. The part on the left is the traditional market of large-scale software applications, and the part on the right represents the infinite market of niche software applications.	3
2.1	A high-level classification of of integration systems based on the type of input and output software artifacts. Mashup tools (gray boxes) integrate data, application logic, and presentation artifacts and produce Web applications with at least a presentation layer (mashups). A Mashup Enabler (ME) scrapes and integrates unpublished data, and publishes them as Web APIs (application logic layer) to be composed by mashup tools.	10
2.2	Mashup tool design space overview.	12
2.3	Live layout classification.	15
2.4	The state-of-the-art mashup tools classified according to expressiveness and usability by non-programmers. An optimal usable mashup tool (gray box), that is missing from the state-of-the-art, is highly usable by non-programmers and provide a medium level of expressive power.	28
3.1	<i>NaturalMash</i> environment: end-users type the recipe of the mashup in the text field and immediately see the output in the visual field. The output contains interactive widgets that can be resized and relocated. The ingredients toolbar helps with API discovery, while the dock gives a summary of the Web APIs (ingredients) used in the current mashup. Ingredients are abstracted away from the technologies they use and are represented as icon.	32
3.2	The grammar of <i>NaturalMash</i> CNL represented in Extended Backus–Naur Form (EBNF).	36
3.3	A meta-model for ingredients (Web APIs).	38

3.4	Wrapper hierarchy.	39
3.5	The meta-model to which parameter schemas conform.	48
3.6	Typing “tweet” results in the autocomplete list showing the labels associated with the Twitter API.	54
3.7	The source of the object (the “map click” label) as well as the object itself (“location”) get highlighted as soon as the cursor is placed in the object text.	55
4.1	The liveness-friendly architecture of <i>NaturalMash</i> aims at supporting live programming by minimizing the response time and providing a mechanism to cope with the service call rate limits. .	58
4.2	The visualized output of Step 1 (linguistic information) for the Listing 3.1 example using the Stanford CoreNLP online tool (http://nlp.stanford.edu:8080/corenlp/). The input is split into two sentences. Part-of-speech tags (VB: base form verb, NNS: noun plural, IN: proposition, DT: determiner, WRB: wh-adverb, NN: noun, VBZ: present verb, VBN: past participle verb) are associated with each word in the input text. Grammatical dependencies (det: determiner, advmod:adverbial modifier, nsubjpass: passive nominal subject, auxpass:passive auxiliary, dobj: direct object, nn: noun compound modifier, prep_about: prepositional modifier) are shown using arrows.	61
4.3	The annotated syntax tree corresponding to the mashup label of Listing 3.1.	62
4.4	The intermediate model generated for Listing 3.1. It contains two control flow graphs: Main Control Flow that corresponds to the imperative sentence “find slides about APIDays”, and Event 1 that is associated with the causal sentence “when a slide is selected, find youtube videos about slide title”. The passing of input data flows to output data flows are represented by ovals.	63

4.5	The generated JOpera visual composition code [94] for Listing 3.1: (a) list of processes created for the mashup: <code>main_control_flow</code> implements the control flow for the imperative sentence, <code>slide_selected</code> implements the control flow associated with the causal sentence (slide select event), and <code>show</code> is the process responsible for creating the user interface of the mashup, (b) control flow implementing Main Control Flow in the intermediate model (Figure 4.4), (c) control flow triggered whenever a slide is selected (it corresponds to Event 1 in the intermediate model), and (d) data flow associated with <code>main_control_flow</code> (“apidays” is a constant passed to the <code>search_slides</code> input parameter).	64
5.1	<i>NaturalMash</i> environment in the first iteration.	69
5.2	<i>NaturalMash</i> environment in the second iteration.	70
5.3	The completion time grows with the complexity of the task at hand. Programmers have a slightly shorter completion time than non-programmers.	74
5.4	The majority of tasks were accomplished successfully (correctly). In terms of accuracy, however, there is no major difference between programmers (P) and non-programmers (NP).	74
6.1	The welcome screen of the evaluation website. It contains a welcome message and a consent form.	82
6.2	This is how <i>NaturalMash</i> is presented to participants in the evaluation website. It contains an Iframe that shows <i>NaturalMash</i> and one auto-hide <i>Task Bar</i> that includes the task description and allows participants to “Finish” or “Skip” the tasks as well as leave a “Message” reflecting what they think while doing the task.	83
6.3	The number of participants recruited by each method (invitation, and crowdsourcing).	84
6.4	Average accuracy of the pre-constructed tasks. P = Programmers, NP = Non-programmers.	88
6.5	Average completion time (in second) for the pre-constructed tasks. P = Programmers, NP = Non-programmers.	89
6.6	Number of “correct” tasks per minutes (efficiency) for programmers, non-programmers, and both (overall).	89
6.7	The participants’ reasons for skipping the open task.	90
6.8	Accuracy of the open tasks for programmers and non programmers as well as overall participants.	91

6.9	Perceived ease-of-use for different features of the system.	92
6.10	Perceived usefulness for different features of the system.	93
6.11	Percentage of all the keystrokes classified as not-permitted.	94
6.12	Top 4 not-permitted keys with their frequency (in percentage). . .	94
6.13	Ratio of using the two discovery methods segmented by the participant groups and task classes.	95
6.14	Difference between programmers and non-programmers in terms of the ratio of using the discovery methods.	95
6.15	The text field should lift the restriction on using Delete keys at the beginning and end of data objects.	98
7.1	A benchmarking framework for mashup tools with four dimensions.	100

Tables

2.1	Summary of the mashup tool design decisions over the mashup design space (part 1). In the “interaction techniques” issue, the big \times and smaller \times represent, respectively, the main and secondary techniques.	22
2.2	Summary of the mashup tool design decisions over the mashup design space (part 2). In the “interaction techniques” issue, the big \times and smaller \times represent, respectively, the main and secondary techniques.	23
2.3	The effect of design decisions on the usability of mashup tools for non-programmers	26
2.4	The effect of design decisions on the expressive power of mashup tools.	27
5.1	The evolution of <i>NaturalMash</i> during the formative user-centered design process in terms of added/removed features. V0, V1, and V2 correspond to the versions of the tool during, respectively, the first, second, and third iterations.	68
6.1	The metrics used in this evaluation to assess the evaluation questions and hypotheses (Section 6.1 on page 77). They are collected either automatically or in a self-reported manner.	79
7.1	Comparison of the state-of-the-art mashup tools in terms of composition model expressive power. The light gray lines highlight mashup tools with more composition expressive power than <i>NaturalMash</i> (highlighted with a dark gray line).	103
7.2	Comparison of the state-of-the-art mashup tools in terms of component model expressive power. The light gray lines highlight mashup tools with (almost) the same component model expressive power as <i>NaturalMash</i> (highlighted with a dark gray line).	104

Chapter 1

Introduction

Hoc and Nguyen-Xuan [59] define *computer programming* as “the process of transforming a mental plan in familiar terms into one compatible with the computer.” In simple words, computer programming is the process of writing source code in a scripting or programming language. According to the Oxford dictionary of English, a *programmer* is “a person who writes computer programs”, and an *end-user* (or simply a *user*) is “the person who actually uses a particular product”, which is, in this case, a computer program.

Based on the programmers’ intent, Ko et al. [70] categorized programming into five types: (i) *professional programming*, that is performed by experienced programmers (e.g., programmers hired by a software company), and the outcome program is intended for a group of end-users, (ii) *expert programming*, that requires highly experienced programmers, and targets not only end-users but also the programmers themselves, (iii) *novice programming*, that is done by inexperienced programmers to serve themselves as well as other end-users, (iv) *computer science students programming*, that is similar to professional programming with the exception that the programmers are inexperienced (fresh graduate students), and finally (v) *End-User Programming (EUP)* that can be done by end-users (programmer or non-programmers) and is primarily intended for the end-users themselves [89]. In this dissertation, we would like to focus on EUP, particularly when it is carried out by non-programmers.

The history of EUP goes way back to early days of computing, when programming (e.g., using punch cards) was mostly intended for the programmers themselves. Later with the enormous growth of software industry, professional programming became mainstream. The rapid increase in the complexity and size of software products and solutions resulted in the emergence of *software development and engineering*, which refer to the systematic cycle of design, im-

plementation (computer programming), and testing. This has also resulted in the emergence of a new profession called *software developer*.

Interestingly, in the U.S. alone, Scaffidi et al. [101] estimated that the number of end-user programmers (i.e., end-users practicing EUP), who primarily use spreadsheets and databases, lies between 12 million and 50 million — compare it with the number of software developers which is approximately 3 million. With the growing number of end-user programmers, EUP is and will remain a vital research area for many years as also predicted by Scaffidi et al. [101].

1.1 The Rise of “Citizen Developers”

The term “citizen developer” was first coined by a 2011 Gartner report¹, and refers to an emerging generation of users who “create new business applications for consumption by others using development and runtime environments sanctioned by corporate IT.” These so-called citizen developers are the same old end-user programmers with a fundamental difference as mentioned in the report: “In the past, end-user application development has typically been limited to single-user or workgroup solutions built with tools like Microsoft Excel and Access. However, today, end users can build departmental, enterprise and even public applications using shared services, fourth-generation language (4GL)-style development platforms and cloud computing services.” In simple words, citizen developers, unlike the traditional end-user programmers, are not limited to spreadsheets and databases, and are expected to build much more sophisticated software applications that might be used by other end-users as well.

The same Gartner report predicted that citizen developers would be building at least a quarter of software applications by 2014. This is not, however, entirely surprising. As we are more and more depending on software technologies, the demand for software developers is dramatically increasing. According to the U.S. Bureau of Labor Statistics², the number of U.S. jobs for software developers is expected to grow by 30% from 2012 to 2020. This is 16% more than the expected job growth across all U.S. occupations over that time. As a result, the fast-growing software industry will fall short of developers. This is where citizen developers come in as vital IT human resources.

But what type of software applications do citizen developers create? Are they expected to create large-scale software applications like Facebook? The answer to the latter is absolutely no. No matter how short the industry is on develop-

¹<https://www.gartner.com/doc/1726747>

²<http://www.bls.gov/>

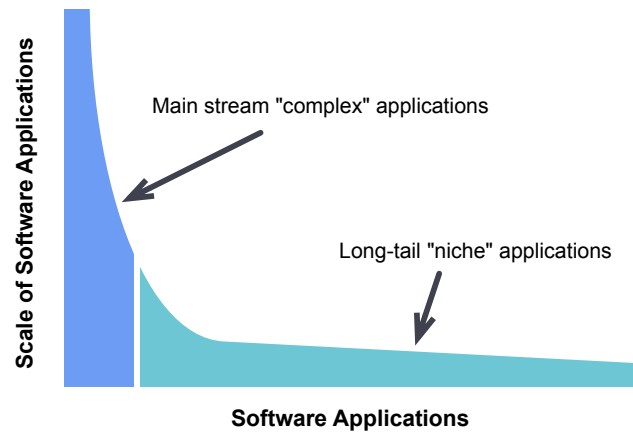


Figure 1.1. The long tail of software applications. The part on the left is the traditional market of large-scale software applications, and the part on the right represents the infinite market of niche software applications.

ers, large-scale applications are to be built by experienced software developers who can take into account critical software quality issues. Answering the former question requires considering the concept of “the long tail” popularized by Chris Anderson [12]. The long-tail of software applications (Figure 1.1) suggests that there exists an infinite market of niche software applications, which is much larger than the traditional market of large-scale software applications. The development of these niche applications is motivated by situational requirements that are transient and apply to a small group of end-users. This marginally exploited market provides a large potential for EUP to be carried out by the emerging citizen developers. Thereby, a considerable software development contributions from citizen developers can be anticipated in the future.

1.2 The Potential of Web Mashups

The proliferation of Web APIs (i.e., reusable software components published on the Web) has turned the Web [93] into a programmable environment. Web mashups (or simply mashups) are a lightweight form of Web applications widely developed and used in this environment [19]. Their development is characterized by the metaphor of hardware components, and is driven by the integration of two or more Web APIs at any of the application layers, i.e., at the data, application logic, and presentation layer [54; 117].

Both in academia and industry, mashup has been a hot topic in recent years. Another Gartner report³ identified mashup as one of the top 10 strategic technologies for 2009. ProgrammableWeb⁴ is a website listing Web APIs and mashups. Since 2006, this website has registered an exponential growth of both Web APIs and mashups. In academia, also, the past 5 years have witnessed an enormous amount of research on mashups [17].

One of the things that makes mashups interesting is their potential to reinforce the vision of “citizen developers”. Mashups are usually lightweight, and thanks to the high level of reuse inherently employed in their development, they guarantee a reasonable level of software quality as well as minimal development costs in terms of time and required skills. Also, the diversity and ever-growing number of Web APIs enable mashups to address a wide range of requirements derived from both established and emerging domains of application [62]. More importantly, mashups run on the Web that is widely considered to be the future platform of all software applications. We believe all these characteristics qualify mashups as the most practical and efficient type of niche software applications.

1.3 Problem Statement

In terms of technical knowledge and skills, there is a spectrum of citizen developers, ranging from non-programmers to experienced programmers. In fact, citizen developers are mainly identified as non-programmers, whose obvious inability to develop software applications (e.g., using general-purpose programming languages) is the main issue hindering the vision of “citizen developers”. This inability does not imply a naive nature for non-programmers, but stems from their lack of motivation to learn and master computer programming. As opposed to software developers who embrace programming as a job, citizen developers are merely driven by personal situational needs that do not outweigh the barriers of learning how to program. This lack of motivation also profoundly concerns programmers as well. They may not be willing to spend days to develop a niche software application from scratch that might be useful for a transient time. The low development cost and high added-value of mashups can be a positive force in this respect. Nonetheless, even the development of mashups faces major barriers that are not by definition mastered by non-programmers. These barriers range from knowing how to reuse Web APIs, to at least a basic understanding of Web technologies and scripting languages (e.g., JavaScript).

³<http://www.gartner.com/newsroom/id/777212>

⁴<http://www.programmableweb.com/>

Hence, citizen developers cannot or are not willing to develop niche software applications unless they are properly and adequately supported to do so.

With a focus on mashups, we formulate our research problem as:

How to support citizen developers (end-users) to develop mashups as niche software applications?

For the sake of convenience, throughout this dissertation, we refer to citizen developers in this context as “end-users” unless we specify otherwise. Since the emphasis is on mashups, the end-users we target are characterized by their active presence on the Web.

1.4 Thesis Statement

To address the stated problem, we consider two particular research fields. The first one is End-User Development (EUD) defined by Lieberman et al. [74] as “a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact.” While EUP entails the intent of programming for self, EUD is concerned with empowering non-programmers to carry out EUP. EUD is an interdisciplinary research field that combines participatory programming [73], meta-design [43], and psychology of programming [23].

The second topic is mashup that is mainly rooted in Web engineering [26]. The research on mashups spans areas including model-driven engineering [41], programming languages [39], and software and service composition [77].

The research scope of this thesis lies at the intersection of EUD and mashups. Within this scope, we aim at developing an EUD system that effectively and efficiently enables end-users to develop mashups.

We assume that:

An EUD system for mashups confronts a trade-off between expressive power and usability by non-programmers. The sweet-spot in this trade-off can be found through providing (i) a proper combination of EUP techniques that ensures both high level abstraction and expressiveness, (ii) metaphors that are familiar and meaningful to everyone, and (iii) an optimal learning experience that supports and develops the skills of non-programmers.

On the basis of the above assumption, we hypothesize the design of an EUD system — that can hit the sweet-spot in the trade-off between expressive power and usability by non-programmers — to adopt:

(i) a hybrid EUP technique combining natural language programming and what-you-see-is-what-you-get, (ii) the “cooking” metaphor in which the “ingredients” (Web APIs) are mixed according to a “recipe” (natural-language source code) for the purpose of preparing “food” (mashup), and (iii) live programming and recommendation techniques to boost optimal learning experience.

We validated the above hypothesis through empirical and conceptual research methods. We designed and implemented the hypothesized EUD system for mashups (*NaturalMash*), and thoroughly evaluated the novel system in the forms of formative, summative, and comparative evaluation. The formative and summative evaluations were done by conducting empirical user studies, and the comparative evaluation was based on a conceptual framework encompassing factors and dimensions that define expressive power of EUD systems for mashups.

1.5 Contributions

In the light of the stated hypothesis, we made the following research and development contributions to the fields of EUD and mashup:

- **A reusable decision space for designing EUD systems for mashups [1; 5; 6].** We conducted a survey of existing EUD systems for mashups with the goal of harvesting their reusable architectural design decisions (see Chapter 2). The decisions are structured into a design space, which (i) helps to classify and explain the heterogeneity of existing systems, and (ii) provides a valuable guidance model to the designers of these systems by enumerating relevant design issues and their dependencies.
- **Natural language programming for mashups [8; 9].** We designed the first natural-language-like programming language for developing the application logic of mashups (see Chapter 3, Section 3.2). The language is highly abstract in a way that it can be understood by non-programmers without prior training. In terms of expressive power, the language allows

to express data flow and data integration tasks as well as various control flow dependencies (e.g., event and sequencing).

- **A specification language for modeling Web APIs [3; 2].** A technical challenge concerning the development of mashups, is the lack of a formal specification language dedicated for describing Web APIs in a unified way. We proposed a JSON-based specification language for modeling and describing various Web APIs (see Chapter 3, Section 3.3). The specification language is expressive, extensible, and highly abstract — these are qualities missing from existing specification languages for Web APIs.
- ***NaturalMash* [8; 9; 4; 7].** We developed *NaturalMash*, a novel EUD system for mashups (see Chapters 3 and 4). Our natural language programming technique is implemented and integrated in a simple What-You-See-Is-What-You-Get editing environment. The novel environment provides guidance and support for editing in the language, allows to discover Web APIs using natural-language searching, and adopts the live programming technique [107] to automate the repetitive task of compiling, deploying, and running.
- **A benchmarking framework for mashups [3; 5].** One of the drawbacks of research in mashups (specially EUD for mashups) is the lack of a systematic approach to compare the state-of-the-art solutions. This shortcoming is largely due to the heterogeneity of existing EUD systems for mashups, making it difficult to arrive at a common ground against which they can be compared. We addressed this shortcoming by proposing a multi-dimensional benchmarking framework (see Chapter 6). We also used the framework to benchmark the state-of-the-art EUD systems for mashups.

1.6 Structure

The rest of the dissertation is structured as follows.

In Chapter 2, we propose a decision space that encompasses a variety of issues and alternatives concerning the design of EUD systems for mashups. The decision space forms a framework based on which we survey the state-of-the-art systems. Also, using the decision space, we classify these systems based on their expected level of expressiveness and usability for non-programmers. Thereby,

we characterize the design of an EUD system that is missing from the state-of-the-art, which, in turn, helps motivate the rest of this dissertation.

In Chapter 3, we explain the main design components of *NaturalMash* intended to fill the identified gap in the state-of-the-art. Using several examples, we outline the grammar, vocabulary, and semantics of the natural programming language, used as the main EUP technique in the system. We describe the JSON-based syntax of the Web API specification language that conforms to an expressive component meta-model. The meta-model not only hides the technological complexity of Web APIs behind a uniform interface but also allows to annotate their capabilities with formal labels. These labels eventually constitute the vocabulary of our natural programming language. Finally, we showcase the intuitive environment of the system by way of a usage scenario.

Next, in Chapter 4, we overview the architecture and implementation of the system. EUD systems for mashups are interactive, and therefore their design involves the notion of perceived response time that is the time an end-user senses from the beginning of input (e.g., editing) and the end of response. Minimizing the response time is a major design challenge that can profoundly enhance the user experience. We propose a high-performance architecture for *NaturalMash* that is set to address this challenge.

We followed a user-centered approach, comprising three iterations of formative evaluation, towards the design of *NaturalMash*. We report the results of the formative evaluations in Chapter 5. We also show how the results of the evaluations gradually drove the design of the system user interface.

To conclude that the system is usable for non-programmers, we conducted a summative evaluation that is thoroughly reported and discussed in Chapter 6. The results showed that non-programmers can effectively and efficiently use the system to create mashups. We also identified a number of usability problems in a formative way sought to be addressed in future work.

In Chapter 7, we get across the multi-dimensional benchmarking framework, using which we also compared *NaturalMash* with the state-of-the-art. Unlike Chapter 2 where we merely determine the expected level of expressive power, in this chapter we actually measure it and compare existing mashup tools accordingly. Combined with the summative evaluation, the comparison results shows that *NaturalMash* is both highly expressive and usable by non-programmers and that we have obtained the desired design.

Finally, in Chapter 8 we remark final conclusions including the limitations of our research as well as the ideas for future work.

Chapter 2

A Survey of the State-of-the-art Mashup Tools

EUD systems for mashups are all consolidated under one single name — *Mashup tools*. The past few years have witnessed many interactive mashup tools, both from research and industry, offering a broad range of characteristics and affordances. Some are based on visual composition languages [67], others feature a high degree of automation and liveness [107], many support collaborative development [46], engaging public and private online communities.

In this chapter, we define mashup tools and survey their state-of-the-art design. The survey is done through the lens of architectural knowledge management [16] with the goal of harvesting reusable architectural design decisions from existing mashup tools. We exploit these decisions to create a unified design space that helps to classify and explain the heterogeneity of the surveyed tools. Finally, we scrutinize the the state-of-the-art by characterizing and collating against a better design for mashup tools.

2.1 Defining Mashup Tools

It has been quite a while since enterprises and software companies realized the benefits of integrating information systems across their organizational boundaries [10]. This integration process can receive (input) and produce (output) software artifacts at any layer of information systems architecture [45], namely presentation, application logic, and data. Based on the input and output software artifacts, Figure 2.1 presents a high-level classification of systems enabling information system integration.

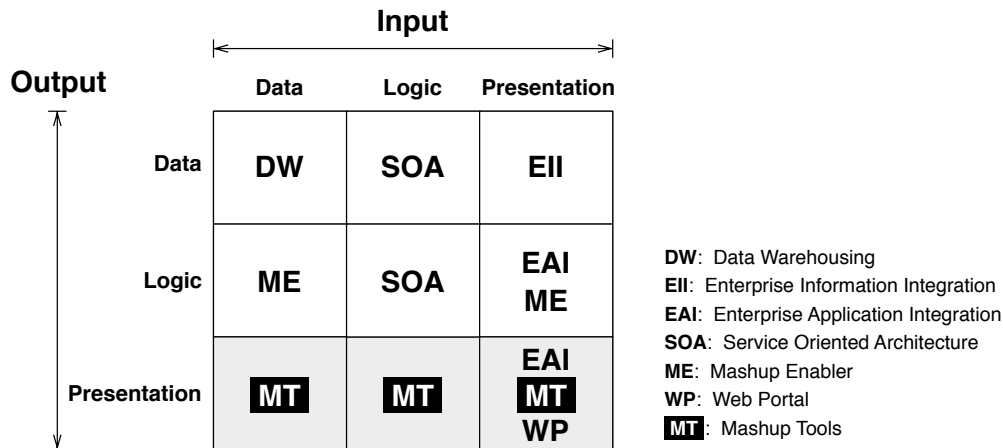


Figure 2.1. A high-level classification of of integration systems based on the type of input and output software artifacts. Mashup tools (gray boxes) integrate data, application logic, and presentation artifacts and produce Web applications with at least a presentation layer (mashups). A Mashup Enabler (ME) scrapes and integrates unpublished data, and publishes them as Web APIs (application logic layer) to be composed by mashup tools.

At the data layer, integrating data from disparate sources across an enterprise can provide powerful business and technical insight throughout the enterprise [53]. Enterprise Information Integration [22] (EII), and Data Warehouse [27] (DW) are the early forms of systems for data integration.

With the help of integration, new functionalities (application logic) can be easily obtained by integrating existing application logics. Service-Oriented Architecture [40] (SOA) streamlines integration at this level by advocating the idea of publishing software applications as services in the cloud with standardized interfaces to be discoverable and understandable by any potential client. These services provide access to not only the application logic of an enterprise application but also its underlying data layer. Consequently, SOA can enable data integration in an easier and more standard way than DW and EII systems.

At a higher layer, even a full-fledged enterprise applications can be developed by putting together existing ones at their presentation layer. Enterprise Application Integration [119] (EAI) was an early attempt in this regard. EAI can also be used to integrate data and application logic extracted from enterprise applications. This is specially useful when the target data and application logic are not published in a database or encapsulates as a standard service. On the Web,

EAI is exploited in Web Portals (WP) that put together one or more Web widgets into a single page website. A popular type of WP is business dashboards that show in a single view different performance indicators relevant to a particular business process (e.g., sales data).

Mashup tools are a new generation of integration systems on the Web. They integrate data, application logic, and presentation to produce Web applications with at least a presentation layer, called mashups. We distinguish Mashup Enablers (ME) that produce Web APIs to be composed by mashup tools. Examples of ME include *Dapper*¹ that allows to scrape websites and publish the results as a data source, and *FeedRinse*² that produces new data sources out of the integration of existing ones.

2.2 Survey Methodology

In order to give a clear structure to the survey, we have constructed a model (Figure 2.2) conforming to a simple, yet powerful decision meta-model proposed in [92] and used the corresponding tool³ to gather and process the knowledge. The meta-model is comprised of design issues and, related to them, design alternatives. The design issues represent a design problem, while each design alternative serves as a potential solution.

Our model was constructed based on the knowledge gained from using/reading about existing mashup tools. In the surveyed literature, we gathered 27 mashup tools, taking into account their availability and design diversity. We further refined the knowledge into design decisions by eliminating overlapping issues and alternatives. Next, we selected 7 top-level design issues and 25 alternative, based on their impact and relevance to usability and expressiveness qualities.

2.3 Design Issues and Alternatives

2.3.1 Design Issue: Automation Degree

A mashup tool needs to leverage automation to lower the barriers of mashup development. The automation degree of a mashup tool refers to how much of the development process can be undertaken by the tool on behalf of its end-

¹<http://open.dapper.net/>

²<http://feedrinse.com/>

³<http://saw.inf.unisi.ch>

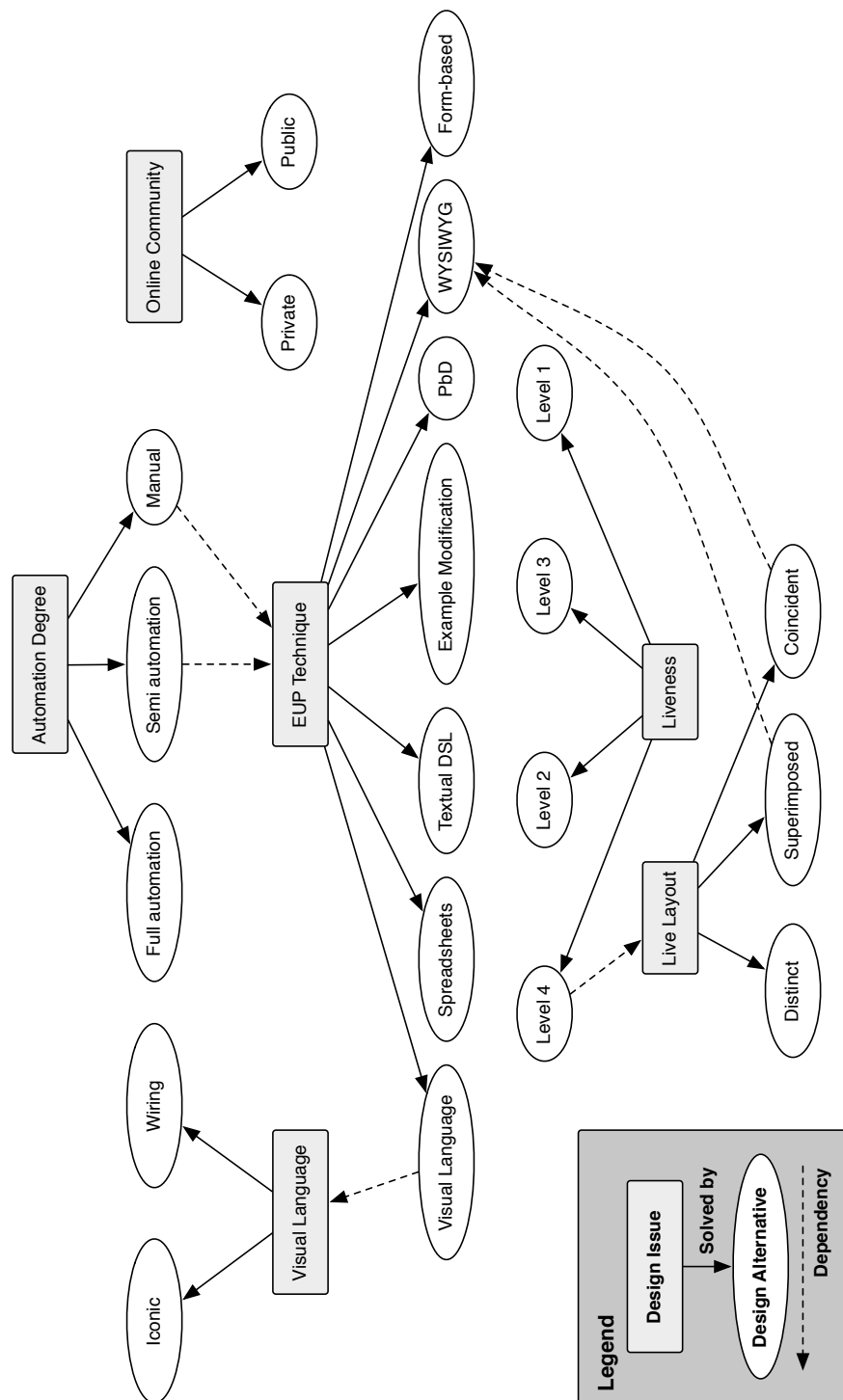


Figure 2.2. Mashup tool design space overview.

users. Considering this, we distinguish between *manual*, *semi-automatic*, and *full automatic* mashup tools.

- **Alternative: Full Automation.** Full automation of mashup development eliminates the need for direct involvement of end-users in the development process. Instead, they will gain a supervisory role with the opportunity to provide input (i.e., requirements) and validate the final result. Since the development process is carried out by the tool, the burden of learning is considerably lifted. Also, if designed properly, it can significantly decrease the effort of mashup development. A great challenge is to ensure the accuracy of a fully automatic mashup tool, so that the (automatically) produced mashups do not deviate from the end-user's needs. The tool may also allow end-users to iteratively validate the resulting mashup and in case of deviation to intervene in the development process. Even though this may partially address the challenge, end-users might encounter the risk of experiencing many time-consuming iterations. As an example, *Piggy Bank* [63] uses semantic technologies to automate the extraction and mixing of content from different websites. It falls back to visual screen scraping techniques, in case the target website does not expose RDF data.

- **Alternative: Semi Automation.** Semi-automatic tools partially automate mashup development by providing guidance and assistance. The role of end-users is to intervene and follow the guidance. The direct involvement of end-users in the development process results in more accuracy (compared with automatic tools) and thus a much lower probability of deviation from their needs. However, the end-users should go through a relatively longer learning curve to be able to create their desired mashups. *ResEval Mash* [64] provides control/-data flow recommendations and guidance. Yet, end-users need to first learn the semantics of the tool and actively select and connect Web APIs on their own to create a mashup.

- **Alternative: Manual.** Manual mashup tools do not offer automation or guidance on the mashup development process. The absence of automation makes the design of the tool relatively easy, but increases the burden of complexity on its end-users. Early generation of mashup tools from academia were, to a large extent, manual. These can be exemplified by *MashArt* [33].

2.3.2 Design Issue: Liveness

In the context of visual languages, Tanimoto proposed the concept of *liveness* [107], according to which four levels of liveness are distinguished. We believe that the applicability of the concept can be found in the domain of mashups as well.

- **Alternative: Level 1. Flowchart as ancillary description.** At the first level, a tool is just used to create prototype mashups that are not directly connected to any kind of runtime system. A prototype mashup usually only has the final user interface without underlying functionality. The goal of mashup tools is to provide a development environment for creating executable mashups. This does not comply with the implications of liveness Level 1, as tools at this level can only help to create non-executable prototype mashups. However, visual tools such as Microsoft Visio can be tailored to create prototype mashups as explained in [115].

- **Alternative: Level 2. Executable flowchart.** At the second level of liveness, a mashup tool only produce some executable blueprints that need to be manually fed to another tool for the purpose of execution. These blueprints are self-contained in terms of documentation, and therefore can serve as reference for end-users. The fact that these blueprints can be automatically transformed into executable mashups implies that they might need to carry on some amount of low-level technical design details, which may make them difficult to interpret by non-programmers. The Enterprise Mashup Markup Language⁴ (EMML), a standard XML-based mashup modeling language from the Open Mashup Alliance (OMA), can be used to produce such blueprints. Even though EMML is not a mashup tool by definition, there are other tools such as *JackBe Presto*⁵, capable of executing EMML blueprints.

- **Alternative: Level 3. Edit triggered updates.** Mashups characterized to the third level of liveness can be rapidly deployed into operation. Deployment in this case can be triggered for example by each edit-change or by an explicit action executed by the developer. The problem at this level is that end-users may be unsure whether the design and runtime environments are in sync with each other, unless they manually press the “run” button, or make use of any other means to trigger the automatic redeployment of the mashup. A good example of a mashup tool at this level is *JOpera* [94]. In the tool design environment, there is a “run” button which automatically executes a mashup and switches the screen to the runtime environment used for debugging and monitoring purposes.

- **Alternative: Level 4. Stream-driven updates.** The fourth level of mashup liveness is obtained by the tools that support live modification of the mashup code, while it is being executed. End-users are allowed to tinker with mashup code in the real time. In turn, changes are (almost) instantly observable, and therefore, quick adaptation is possible. As a result, the development

⁴<http://www.openmashup.org/>

⁵<http://www.jackbe.com/enterprise-mashup/>

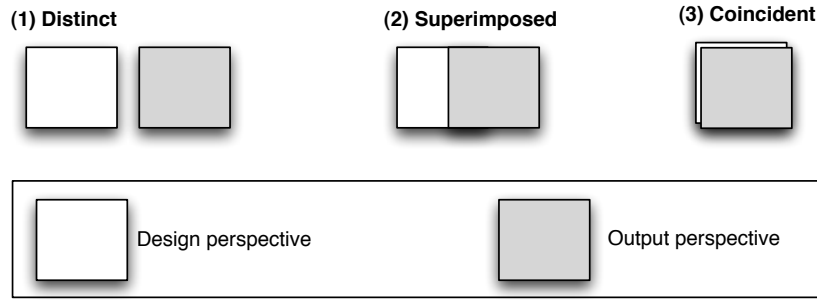


Figure 2.3. Live layout classification.

cycle is very rapid. High design agility comes with the risk that uncontrolled changes to an operational system could make it fail. The same danger applies in case of live collaboration on mashup development that can potentially leave the system inconsistent. Finally, an important challenge is that highly responsive environments can result in high costs of running the mashup, as – for example – remote Web APIs need to be invoked every time a change is done on the mashup code. This can become problematic when, for instance, the Web API has a call rate limit. *DashMash* [25], *Omelette* [28], and *RoofTop* [61] support liveness at Level 4 by merging the mashup design and runtime environments, and proving a mechanism to keep both of them synchronized.

2.3.3 Design Issue: Live Layout

At Level 4 of liveness, we distinguish three variations of mashup tools: distinct, coincident, and superimposed (Figure 2.3). The classification is made based on how design and output perspectives are positioned in a mashup tool environment. Design perspective is where the end-user creates or manipulates a mashup. Likewise, output perspective constitutes the result of the run-time execution of the mashup being created or modified in the design perspective.

- **Alternative: Distinct.** This type of live mashup tools separates the design and output perspectives, in a way that the two perspectives are recognizable from each other. It should be noted that the expected output perspective might be integrated into the same user interface as the design perspective or not. The former case has the advantage of allowing the tool environment to be more self-contained. In the latter case, however, the output perspective resides in the actual target environment, in which the output mashups are to be deployed and executed. The output perspective is thus remotely connected to the design perspective that is the main part of the mashup tool environment. For instance,

the output perspective may be integrated with a Web browser (e.g., FireFox) that gets connected to the mashup tool environment using a plug-in. The advantage of a realistic output perspective is that it shows not only the final execution of the mashup being created, but also its final deployment environment. Still, this separation imposes more barriers on the end-users' side, as they need to be able to distinguish and switch between the two perspectives. This is specially the case when the output perspective is remotely connected to the tool environment. The *distinct* variation is exemplified by *MashStudio* [116], in which the environment interface only consists of a visual editor (design perspective) that is remotely connected to the output perspective (a mobile client).

- **Alternative: Coincident.** This variation comes in effect when the design and output perspectives of the mashup tool environment completely fold over each other, in a way that they are not distinguishable anymore. This is synonym to What You See Is What You Get (WYSIWYG), where the interface shown to the end-user allows to see and manipulate content (e.g., graphical object, text, etc.) that is very similar to the output mashup begin created. The main advantage of this variation is the direct manipulation feedback governed by the WYSIWYG interface. From the mashup tool design perspective, Its major shortcoming is the inability of choosing other interaction techniques than WYSIWYG for mashup programming. WYSIWYG interfaces since they mainly target the user interface of the mashup are unable to provide adequate expressive power for manipulations of the business logic and data integration layers of the mashup being created. *Intel MashMaker* [38] is an example of this type of mashup tools.

- **Alternative: Superimposed.** In this type, part of, or the whole of, the output perspective is superimposed with the design perspective, but does not cover it completely. The output perspective is basically based on WYSIWYG (similar to the *coincident* variation) that has the advantage of allowing direct manipulation. In contrast to the *coincident* variation, the design perspective is not limited to the WYSIWYG part (the output perspective). This allows to use other interaction techniques in the design perspective to further compensate the shortcoming of WYSIWYG in business logic creation. Analogous to the *distinct* live mashup tools, the output can be integrated or remotely connected. In other words, *superimposed* live mashup tools take the best of the other two variations (*coincident* and *distinct*) and apply them to their own shortcomings. Spreadsheet-based mashup tools, if implemented as a live tool, can be included in this variation.

2.3.4 Design Issue: Online Community

Online communities are an important resource in assisting end-users to program [89]. They can potentially support technical discussion as well as collaborative mashup categorization, sharing, rating, and recommendation [48]. An online community can take the form of a blog, a newsgroup, a chat room, or even a social network, depending on the role it is supposed to fulfill. From a security and privacy point of view, currently available online communities for mashup tools fall into two distinct types: *public*, and *private*.

- **Alternative: Public.** The content published in public communities are accessible by any end-user on the Web who wishes to join them. This, however, does not imply that these communities do not require registration prior to accessing them. The added value of a public community lies in its great potential for growth, ultimately resulting in the increased number of the tool end-users. As the content shared in the community is public, end-users may refuse or refrain from sharing certain details. *Yahoo! Pipes*⁶ maintains one of the largest public communities of mashup developers. Members of the community can share, discuss, reuse, and categorize mashups created with the Yahoo! Pipes tool.

- **Alternative: Private.** The authority to join a private or a gated community is granted on the basis of compliance with some special criteria. These criteria can be having an invitation or being a registered member of a certain organization. Private communities are usually small in number of end-users. The content stored in private communities is inaccessible to non-members, resulting in a higher level of confidence for end-users to discuss issues related to their organization. These communities require much more effort to start. Content should be mostly created by the community staff, since with a small number of end-users, there will not be much user-generated content initially. *IBM Mashup Center (IMC)*⁷ allows enterprises to build their own private community, organized around a centralized catalog. End-users can publish mashups to this catalog so that others can discover and reuse them.

2.3.5 Design Issue: End-User Programming Technique

There have been a number of interaction techniques through the use of which the barriers of programming can be lifted from end-users [87]. We list below some representative techniques which have been used by mashup tools. Some tools are known for using multiple techniques in combination.

⁶<http://pipes.yahoo.com/>

⁷<http://www.ibm.com/software/info/mashup-center/>

- **Alternative: Spreadsheets.** Spreadsheets are one of the most popular and widely used end-user programming approaches to store, manipulate, and display complex data. The advantage of using spreadsheets for creating mashups lies in their ease-of-use, intuitiveness, and expressive power to represent and manage complex data ([58]). The main shortcoming of spreadsheets-based tools is the lack of support for designing the mashup UI. *Mashroom* ([112]) adapts the idea of spreadsheets and adds the nested tables to support complex data formats such as XML and JSON. *Husky*⁸ is also another spreadsheet-based tool aiming at streamlining service composition. *Vegemite* [75] extends the *CoScripter* web automation tool [76] with a spreadsheet-like environment

- **Alternative: Programming by Demonstration (PbD).** As opposed to direct programming, Programming by Demonstration (PbD) suggests to teach a computer by example how to accomplish a particular task [30]. This is a powerful technique that helps remove much of programming barriers. End-users demonstrate what is the mashup they want without worrying about how it should be programmatically implemented. Termination conditions and branches are two important artifacts in the design of a mashup control flow graph—a graph that defines the execution order of components and statements. These artifacts are not, however, feasible to be directly articulated by PbD technique [89]. *Karma* [109], for example, allows extract, filter, and aggregate content on the Web through demonstration ([102]). *Intel MashMaker* utilizes PbD to extract, store, manage, and integrate data from the Websites being browsed by the end-user.

- **Alternative: Visual Language.** A visual programming language, as opposed to a textual programming language, is any programming language that uses visual symbols, syntax, and semantics [86]. If designed properly, a visual language offers a high level of abstraction, thus better targeting the needs of end-users. One of their strengths is their ability to support more than one view at the same time, e.g. showing both the design and runtime environments in the same screen. A potential challenge is to make the most of the available screen space (i.e., visual scalability), as the ability to layout diagrams in two dimensions can be outweighed by the complexity and the size of the diagrams. *SABRE* [81] is based on a visual language corresponding to Reo [13], a coordination language that is used to define the logic of the mashup.

- **Alternative: Textual Domain-Specific Language (DSL).** Textual Domain Specific Languages (DSL) are small languages targeted for solving certain problems in a specific domain ([110]). DSLs can also be used as a EUP technique for

⁸<http://www.husky.fer.hr/>

reducing programming efforts ([97]). Textual DSLs define textual syntax, that may or may not resemble an existing general-purpose programming language. DSLs, particularly those built internally on top of a general purpose programming language, usually offer a high expressive power. In terms of learning barriers, textual DSLs are similar to programming languages, and still cannot be used by non-programmers [30]. *Swashup* offers a textual DSL for mashups, based on Ruby-on-Rails. *WMSL* exemplifies a DSL having its own paradigm and syntax.

- **Alternative: What-You-See-Is-What-You-Get.** WYSIWYG enables end-users to create and modify a mashup on a graphical user interface which is similar to the one that will appear when the mashup runs. Since end-users always see the resulting mashup, the whole development process is streamlined. Another potential benefit is the increase of the tool directness. End-users place visual objects exactly in the places where they are meant to be. However, the application logic of a mashup such as data filtering and conversion happens in the backend where is not visible in the graphical user interface, and therefore is not directly accessible for modification using a pure WYSIWYG tool. *DashMash* is a WYSIWYG tool. End-users can drag-drop-and-connect a set of boxes (widgets) whose current visual positions are the same both in the design time and runtime.

- **Alternative: Form-based.** In form-based interaction, end-users are asked to fill out a form to create a new or change the behavior of an existing object. Filling out online forms has nowadays become an ordinary task for end-users on the Web. This can be interpreted as a proof for “naturalness” of form-based tools [108]. However, form-based tools cannot handle complex composition patterns for mashups [66]. *ServFace Builder* [90] allows to customize user interface and data sources using a form.

- **Alternative: Programming by Example Modification.** Another powerful technique to remove the burden of programming is to let end-users modify and change the behavior of existing examples, instead of programming from scratch [79]. Provided that adequate mashup examples are available, in most cases the modification of a mashup example or the customization of a predefined mashup template requires a small effort. Searching for appropriate example as a suitable starting point for the work is a challenging task for non-programmers, as they are not familiar with any programming languages. With the ever increasing number of Web APIs, providing adequate mashup examples derived from all possible combinations of these APIs is not feasible. *d.mix* [55] allows to sample elements of a website, and then generates the corresponding source code producing the selected elements. These source codes are stored in a repository, where they can be discovered and edited by end-users.

- **Alternative: Natural Language Programming.** *Open Mashup* ([18]) incorporates natural language programming for composing mashups on mobile devices. Even though natural language is considered a natural way for humans to command computers [49; 56; 68] it cannot be efficiently used for user interface integration and design which are integral part of mashup development [118]. In the area of personal information management, *Automate* [18] uses a simplified CNL for context-sensitive personal automation. Another similar system is *IFTTT*⁹ which, even though it is based on natural language, restricts the end-user's input using a structured visual editor. Also, *IFTTT* only allows to create mashups based on a specific control-flow pattern (if this then that) using a predefined list of components

2.3.6 Design Issue: Visual Language

Visual programming languages proposed by existing mashup tools fall into two main classes. The first class contains the tools that are based on a *visual wiring language*. The second consists of those incorporating an *iconic visual language*.

- **Alternative: Wiring.** In a visual wiring language for mashups, activities are visualized as solid or form-based boxes that can be wired to each other. Each activity can represent a mashup component or a predefined operation like filtering, sorting, and merging. Wires indicate the connection between these activities. In the realm of service composition, wiring languages can be considered one of the most *explicit* and popular approaches to express a composite service, due to the one-to-one relationship between the flow of control and data from one activity to another and visual boxes wired to each other. Wiring languages can cause readability problems, when there are multiple crossing edges, or when the visual graphs exceed the screen size. In the latter case, it is essential for a tool to provide auto-layout features. The visual language incorporated by *MashArt* represents queries and processing tasks over data sources as form-based boxes, which are able to connect to each other.

- **Alternative: Iconic.** An iconic visual language represents objects to be handled by the language as graphical icons. Sentences are made with one or more icons that are related to each other according to a predefined syntax. Properly designed icons are generally easily interpreted, understood and remembered by end-users. An iconic visual language requires to invest significant effort and thought into icon design [71]. This is essential to avoid any further changes to the appearance of the icons, which causes confusion due to unex-

⁹<https://ifttt.com>

pected behavior. *VikiBuilder* [57] enables generation of visual wiki instances by combining various data sources. The tool uses iconic annotations to represent various predefined entities like adapter, data source, and semantic extractor.

2.3.7 Design Issue: Design Distance

The user interface of a mashup tool contains presentation objects (e.g., buttons, icons representing components, and domain-specific symbols) that abstract their underlying implementation in a programming language. The gap between presentation objects and their underlying implementation is called design distance [85]. A long design distance in a mashup tool results in the increase of its usability (the objects will be closer to the end-user's way of thinking). Even though usability here has the upper hand, a long design distance may potentially endanger expressiveness. To cope with this, [84] proposes three techniques: customization, integration, and extension. When combined together, these techniques (from left to right) enable a gradual shortening of the design distance to ensure both usability and expressiveness.

- **Alternative: Customization.** In mashup tools, customization shortens the design distance by allowing to modify the functionality or user interface of an existing mashup through parameterization. Many mashup tools, such as *MyCocktail*¹⁰, use this technique to enable the alteration of Web API calls or modification of the size and position of widgets in a mashup user interface.

- **Alternative: Integration.** This technique allows to create new mashups out of the composition of available Web APIs. This technique considerably shortens the design distance but still relies on the Web APIs available in the tool library. Most mashup tools, exemplified by *CRUISE* [96], accommodate this technique as an integral part of their design.

- **Alternative: Extension.** This technique provides the shortest design distance and allows to extend a mashup tool by adding new Web API to its library. To accommodate this technique, the mashup tool needs to possess an open Web API library, to which end-users can contribute new Web APIs. For instance, *JackBe Presto* offers an open Web API library that can be extended by its end-users.

¹⁰<http://www.ict-romulus.eu/web/mycocktail>

Design issues and alternatives	Mashroom	Husky	Karma	MashMaker	Vegemite	Yahoo! Pipes	IMC	JOpera	JackBe Presto	Marmite	MashArt	ResEval Mash	SABRE	MashStudio
Automation degree														
Manual	×	×			×						×			
Semi-automatic			×	×		×	×	×	×	×		×	×	×
Full automatic														
Liveness														
Level 1														
Level 2														
Level 3	×	×	×	×	×	×	×	×	×	×	×	×	×	
Level 4														×
Live Layout														
Distinct														×
Superimposed														
Coincide														
Online communities														
Private							×							
Public						×			×					
EUP techniques														
Spreadsheets	×	×	×	×	×					×				
PbD			×	×										
Visual language						×	×	×	×	×	×	×	×	×
DSL														
WYSIWYG	×			×	×		×		×					
Form-based						×	×	×	×	×	×	×		×
Example modification														
NLP					×									
Visual language														
Iconic													×	
Wiring						×	×	×	×	×	×	×		×
Design distance														
Customization	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Integration	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Extension						×	×	×	×		×	×		

Table 2.1. Summary of the mashup tool design decisions over the mashup design space (part 1). In the “interaction techniques” issue, the big × and smaller × represent, respectively, the main and secondary techniques.

Design issues and alternatives	MyCocktail	MashableLogic	Swashup	WMSL	ServFace	DashMash	Omelette	CRUISE	RoofTop	d.mix	Open Mashup	Piggy Bank	IFTT
Automation degree													
Manual			×	×									
Semi-automatic	×	×			×	×	×	×	×	×	×		×
Full automatic												×	
Liveness													
Level 1													
Level 2													
Level 3	×	×	×	×	×				×	×	×	×	×
Level 4						×	×	×	×				
Live Layout													
Distinct													
Superimposed													
Coincide						×	×	×	×				
Online communities													
Private													
Public	×	×											×
EUP techniques													
Spreadsheets												×	
PbD													
Visual language		×											
DSL			×	×									
WYSIWYG	×	×			×		×	×	×				
Form-based	×	×			×	×	×	×	×	×		×	×
Example modification										×			
NLP											×		×
Visual language													
Iconic													
Wiring		×											
Design distance													
Customization	×	×	×	×	×	×	×	×	×	×	×	×	×
Integration	×	×	×	×	×	×	×	×	×	×	×		×
Extension			×	×		×	×	×		×			

Table 2.2. Summary of the mashup tool design decisions over the mashup design space (part 2). In the “interaction techniques” issue, the big × and smaller × represent, respectively, the main and secondary techniques.

2.4 Survey Summary

Tables 2.1 and 2.2 summarize the survey of the design of 27 mashup tools based on the proposed decision space. As it can be seen in the tables, the majority of the surveyed mashup tools are semi-automatic. Full automation is rarely an option (only Piggy Bank) mostly due to the technological complexity and heterogeneity of mashup development.

Interestingly, the vast majority of the tools already support liveness at Level 3 through a “run” button that takes the end-users to the runtime environment where the mashup will be deployed and executed. Few recent mashup tools (*Omelette*, *RoofTop*, *DashMash*, and *MashStudio*), however, support Level 4, mostly in a coincident layout form.

Establishing an online community, despite its benefits [113], is not a popular design decision. This is probably due to the fact that most of the surveyed mashup tools are research prototypes that have not been yet deployed for real use. Therefore, only some industrial mashup tools (e.g., Yahoo! Pipes, and IBM Mashup Center) provide an online community for their end-users.

According to the tables, the most popular EUP techniques are visual wiring language programming and form-based interaction. When these two techniques are used together in one mashup tool, the visual wiring language commonly consists of graphical boxes that represent data sources or processing operators, and contain a form for customization. The combination of these two techniques is in fact the old software composition metaphor [14] that dates back to before the invention of mashups. As a result, other techniques such as NLP and programming by example modifications are not that common.

Regarding the design distance issue, all of the semi-automatic and manual tools support integration and customization techniques. Enabling extension requires an open architecture that is not widely adopted by these tools.

2.5 The Need for Another Mashup Tool

Mashup tools must possess two important qualities: expressiveness, and high usability for non-programmers. It is the top priority for a mashup tool to be highly usable by non-programmers, as they are the dominant population of end-users. Expressiveness is also important as it enables end-users to develop sophisticated, feature-rich mashups. However, the interpretation of “useful” expressiveness may differ from one mashup tool to another depending on the application domain they target [65]. For instance, a mashup tool designed for research per-

formance evaluation [64] is considered expressive as long as it allows end-users to accomplish the required tasks (e.g., calculating a h-index score). Since there is a long-tail of application domains for mashups, in our thesis, we favor the highest level of expressiveness as possible for the design of mashup tools. This is due to the fact that a highly expressive mashup tool can be potentially useful in any application domain, assuming that its high usability for non-programmers is assured and not compromised.

In this section, we illustrate the trade-off between the two qualities of mashup tools (expressiveness, and usability for non-programmers). To this end, we first discuss how these qualities are affected by the design issues mentioned in this chapter, and then analyze and classify the surveyed mashup tools accordingly. In doing so, we also conceptualize the design of a desired mashup tool as the sweet-spot in the trade-off, and show that it is missing from the surveyed state-of-the-art.

The main design issues that affect expressiveness (Table 2.4) include design distance, EUP techniques, and automation. Likewise, the design issues determining whether a tool is usable by a non-programmer (Table 2.3), are liveness, live layout, EUP techniques, automation, and online community.

The three techniques to shorten the design distance (customization, integration, and extension), when properly implemented within a mashup tool, can significantly increase its expressive power. As mentioned previously, most of mashup tools support the first two techniques (customization and integration), and only a handful of them employ the three techniques together. Supporting extension technique is important as the number of new Web APIs providing interesting functionality and data is rapidly increasing.

EUP techniques also each allow up to a certain level of expressiveness. For instance, whereas textual DSLs are capable of offering a high level of expressive power, the form-based technique limits the user's interaction to customization tasks. To be specific, textual DSL, visual wiring language programming, and natural language programming allow the highest level of expressive power. Moreover, there is an association (Figure 2.2) between EUP techniques and the techniques to minimize the design distance. For instance, form-based, visual language programming, and textual DSLs can be respectively associated to customization, integration, and extension techniques. Additionally, EUP techniques have different level of abstraction, which in turn have impacts on usability. Highly abstract techniques such as PbD, Form-based, and WYSIWYG, are more suitable for non-programmers. There is, therefore, a trade-off here between the levels abstraction and expressive power: higher level of abstraction compromises expressiveness. Nevertheless, the metaphor of the EUP technique

Design issues and alternatives	Low	Medium	High
Automation degree			
Manual	×		
Semi-automatic		×	
Full automatic			×
Liveness			
Level 1	×		
Level 2	×		
Level 3		×	
Level 4			×
Live Layout			
Distinct		×	
Superimposed			×
Coincide			×
Online communities			
Private			×
Public			×
EUP techniques			
Spreadsheets		×	×
PbD			×
Visual language	×	×	
DSL	×		
WYSIWYG			×
Form-based			×
Example modification	×	×	
NLP		×	×

Table 2.3. The effect of design decisions on the usability of mashup tools for non-programmers

Design issues and alternatives	Low	Medium	High
Automation degree			
Manual			×
Semi-automatic		×	
Full automatic	×		
Design distance			
Customization	×		
Integration		×	
Extension			×
Online communities			
Private			×
Public			×
EUP techniques			
Spreadsheets	×	×	
PbD	×		
Visual language		×	×
DSL			×
WYSIWYG	×		
Form-based	×		
Example modification		×	×
NLP		×	

Table 2.4. The effect of design decisions on the expressive power of mashup tools.

Usability by non-programmers ↑				
		Highly Usable	Optimal Usable	Ideal
		<i>DashMash</i> <i>ResEval Mash</i> <i>Piggy Bank</i> <i>CRUISE</i> <i>Open Mashup</i> <i>IFTTT</i> <i>Omelette</i>		
		Moderately Usable	Moderate	Optimal Expressive
	Medium	<i>Yahoo! Pipes</i> <i>Vegemite</i> <i>ServFace</i> <i>Husky</i> <i>Karma</i> <i>Marmite</i> <i>RoofTop</i>	<i>MashArt</i> <i>JackBe Presto</i> <i>IMC</i> <i>MashStudio</i> <i>SABRE</i> <i>Mashroom</i>	<i>d.mix</i> <i>JOpera</i>
	Low	Bad Design	Moderately Expressive	Highly Expressive
			<i>MashMaker</i> <i>MashableLogic</i> <i>myCocktail</i>	<i>WMSL</i> <i>Swashup</i>
		Low	Medium	High
		Expressive power →		

Figure 2.4. The state-of-the-art mashup tools classified according to expressiveness and usability by non-programmers. An optimal usable mashup tool (gray box), that is missing from the state-of-the-art, is highly usable by non-programmers and provide a medium level of expressive power.

is an influential factor on usability as well. In spite of the popularity of visual languages based on the wiring paradigm, according to a study conducted by Namoun et al. [88], these languages in the context of mashups are not “natural” to many non-programmers. In other words, a diagram representing the flow of data and control is more of a metaphor suitable for programmers rather than non-programmers.

Automation is another issue that directly affects both expressiveness and usability. The increase of automation results in the decrease of expressiveness. The more tasks are automated by the tool, the less opportunities end-users have to express and communicate their ideas. In the same way, more automation is commonly believed to better serve non-programmers [44]. In case of full automation, however, the challenge lies in how to support end-users to communicate their requirements (so the system will take of the rest) and give guidance and feedback if automation fails and falls back. The trade-off here is that increasing automation results in decreasing expressive power, while having the opposite effect on usability.

Liveness is an important issue affecting the usability of a tool. The higher the level of liveness, the better cognitive support for non-programmers. Level 3, which is supported by most of mashup tools, still requires end-users to distinguish between the design-time modeling and composition environment and its run-time version, where the mashup execution occurs. On the other hand, liveness at Level 4, which is not supported by most mashup tools, helps non-programmers to gradually develop their skills in using the tool. Accomplishing and then immediately validating small steps one at a time towards solving a bigger problem lead the end-users towards a gradual and gentle learning process. Moreover, the best way to implement liveness at Level 4 is based on superimposed layout. This way, due to the possibility of using another EUP technique, much more expressive power can be added to the tool.

Empowering end-user communities is of value and can improve usability [89]. They can be used to promote sharing of mashups, technical discussion, and collaborative categorization [48]. One of the most promising approaches to lower cognitive costs and increase motivation is to facilitate collaborative development [43]. To this end, establishing an online community is crucial as it brings together users with similar interests and common ground [113]. It should also be noted that the full potential of an online community is exploited only when it is internally built upon the actual end-users of the tool (like in *Yahoo! Pipes* and *JackBe Presto*), not externally in the form of a technical blog or a fan page (like in *MashableLogic*). Online communities and collaborative development are fascinating topics, which, however, will not be explored in this dissertation.

By considering the discussed impacts of the above design issues on expressiveness and usability for non-programmers, as well as based on what decisions the surveyed mashup tools have made about these issues, we roughly classify these tools into one of the following groups (Figure 2.4): *highly expressive*, *highly usable*, *moderately expressive*, *moderately usable*, *moderate*, *optimal expressive*, *optimal usable*, *ideal*, and *bad designed tools*. Clearly, high expressive tools are domain-specific scripting and programming languages designed for mashup development such as *Swashup* and *WMSL*. Almost all WYSIWYG tools, such as *DashMash* and *Omelette*, are classified as highly or moderately usable, due to their high level of abstraction and direct manipulation. Mashup tools based on visual wiring languages (e.g., *JackBe Presto*, and *IBM Mashup Center*) mostly fall into the moderate and moderately expressive groups. Both ideal and bad designed groups are empty, with a difference that it is possible to have a bad designed mashup tool, whereas an ideal tool is implausible to achieve. This is due to the mentioned trade-offs within and between the design issues (e.g., automation and EUP techniques issues) making it impossible to reach the highest levels of both expressiveness and usability for non-programmers within the design of one mashup tool. We also introduced two optimal groups — one focusing on expressiveness and the other on usability. Since usability for non-programmer has the upper hand, we opt for the development of an optimal usable mashup tool, which is missing from the state-of-the-art. Such a tool hits the sweet-spot in the trade-off between expressiveness and usability for non-programmers.

Chapter 3

NaturalMash: A Live Natural Mashup Tool

In Chapter 2, we reviewed the state-of-the-art mashup tools and identified the need for an “optimal usable” one, and defined it as an EUD system that, given the trade-off, provides the highest level of usability by non-programmers and a moderate level of expressive power. In this chapter, we explain the design of *NaturalMash* that we claim is an “optimal usable” mashup tool. We first outline the top-level design decisions (Section 3.1), and then move on to describe the core components of the system including, natural language programming (Section 3.2), Web API specification modeling language (Section 3.3), and the development environment (Section 3.5).

3.1 Design Decisions

The design of *NaturalMash* requires a novel hybrid EUP techniques ensuring both high level abstraction and expressiveness. Moreover, it is essential that the metaphors used in the design are familiar and meaningful to a wide range of end-users. Above all, the design should be able to provide an optimal learning experience [31] that supports and develops the skills of non-programmers. Accordingly, we made the following top-level design decisions:

- **Semi-automated mashup development (D1):** According to the trade-off between automation on one side and expressive power and usability on the other side, we decided to design *NaturalMash* as a semi-automatic tool. Thereby, while the development of mashups should be carried out by end-users, partial automation is provided through recommendation.

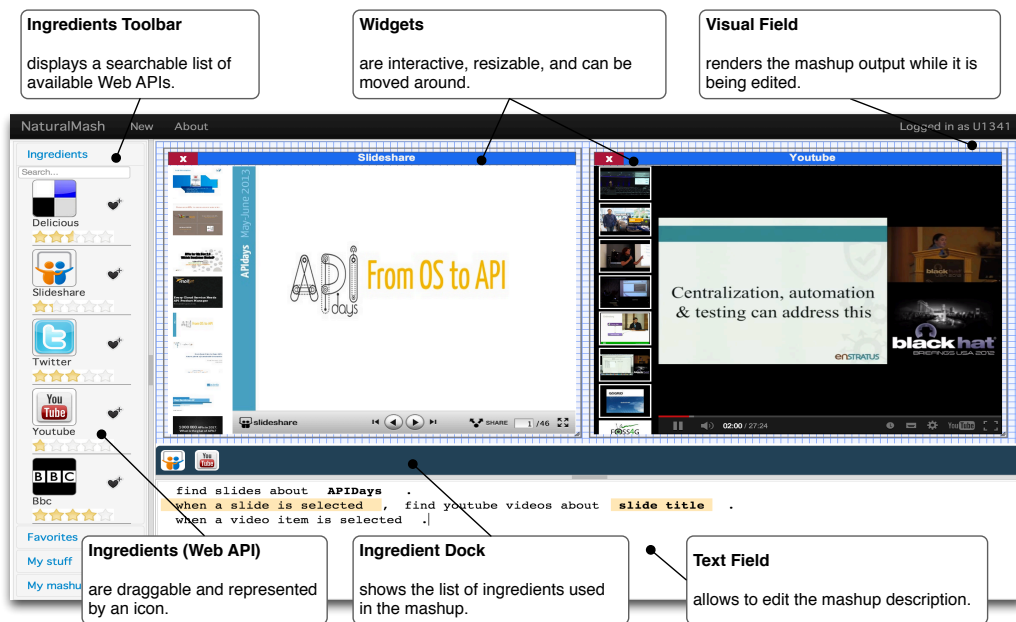


Figure 3.1. NaturalMash environment: end-users type the recipe of the mashup in the text field and immediately see the output in the visual field. The output contains interactive widgets that can be resized and relocated. The ingredients toolbar helps with API discovery, while the dock gives a summary of the Web APIs (ingredients) used in the current mashup. Ingredients are abstracted away from the technologies they use and are represented as icon.

- **Natural language programming combined with WYSIWYG (D2):** Natural language programming as an end-user programming technique can potentially provide a good level of expressive power. Also, natural languages (e.g., English) are readily understandable by their speakers. In the design of *NaturalMash*, natural language programming is enabled through a *Controlled Natural Language* (CNL) — a subset of a natural language (e.g., English) restricted in terms of vocabulary and grammar. The reason for using a CNL is to ensure the accuracy of the system compiler. From the expressive power point of view, the *NaturalMash* CNL empowers end-users to describe relatively complex process orchestration and data integration logic as well as the composition of widgets (all at a very abstract level).

While natural language programming is intuitive and can give a high level of expressive power for developing the mashup back-end, it cannot be effectively and efficiently used for the front-end (user interface) design. On the other hand, WYSIWYG provides intuitive direct manipulation facilities for the front-end design — while it falls short of providing adequate expressive power for developing the back-end.

The reason for selecting and combining these techniques (natural language programming and WYSIWYG) lies in the fact that they augment one another's strengths and compensate for one another's weaknesses. To the best of our knowledge, *NaturalMash* is the only mashup tool that adopted this novel hybrid EUP technique.

- **Superimposed live programming (D3):** *NaturalMash* incorporates the live programming paradigm [29] (liveness level 4). As a result, end-users can more easily bridge the gulf of evaluation (the degree of difficulty of assessing and understanding the state of the system [91]). This in turn leads towards an optimal learning experience [98].

We also adopted a superimposed live layout, in which the design (WYSIWYG) and output perspective are partially overlapped. The overlapped area is the output/design perspective consisting of the WYSIWYG interface, and the disinterested area the design perspective powered by natural language programming. As it was mentioned previously, the advantage of the disinterested perspective is achieving added expressive power.

- **“Cooking” Metaphor (D4):** Many end-users, specially non-programmers, might not be familiar with the technical terminology related to mashups such as service composition and Web APIs [88]. In order to bridge this gap, we designed *NaturalMash* based on the familiar metaphor of *cooking*, according to which Web APIs are referred to as “ingredients”, and the mashup source code (in natural language) is the “recipe” to mix these ingredients.

- **Simple Graphical User Interface (D5):** A simple yet powerful graphical user interface can potentially reduce the learning barriers and make the system more intuitive. Hence, we designed *NaturalMash* (Figure 3.1) to have a Single Page Application (SAP) interface composed of merely four main components that together control all the functions of the system: (i) *text field* providing advanced support for the natural language programming, (ii) *visual field* implementing the PbD WYSIWYG interface for both the design and preview of the user interface of the mashup being created, (iii) *ingredient dock* graphically representing the Web APIs used by the mashup, and (iv) *ingredients toolbar* containing all the mashups created by the end-users and a searchable list of Web APIs.

3.2 NaturalMash Controlled Natural Language

The *NaturalMash* CNL is an abstract, executable language for modeling the development of mashups at all the layers of presentation, application logic, and data. We first introduce the language with a few examples as follows.

Listing 3.1 is the recipe (executable text written in the CNL) of a mashup that searches Slideshare¹ (it is a Website for sharing and finding presentations and documents) for a topic or event (in this example “APIDays”), and then uses the title of each resulting slide to accurately search for its corresponding presentation video in YouTube².

```
Find slides about APIDays. When an item is selected,
find YouTube videos about slide title.
```

Listing 3.1. A presentation search mashup.

Listing 3.2 is an example recipe of a map-based mashup. The mashup includes a user interface composed of a Google Maps widget³ and an HTML table widget. The content of the table displays a stream that aggregates content from the BBC News⁴ and CNN News feeds⁵. When a news item in the table is selected, the Yahoo! Placemaker service⁶ extracts geographical data (e.g, longitude and

¹<http://www.slideshare.net/developers>

²<https://developers.google.com/youtube/>

³<https://developers.google.com/maps/>

⁴<http://www.bbc.co.uk/news/10628494>

⁵<http://rss.cnn.com/rss/edition.rss>

⁶<http://developer.yahoo.com/geo/placemaker/>

latitude) from the text. The geographical data is used to place a marker representing the news item on the map.

```
Combine BBC Top News with CNN Top News.
When an item is selected, extract the location of the item,
and place a marker on the map.
```

Listing 3.2. A location-based news feed mashup.

Listing 3.3 builds a mashup combining Twitter⁷, the YouTube player, a HTML table, and a regular expression component, extracting values from an input using a set of predefined patterns (e.g., “YouTube video link”, “Flickr image link”, etc.). It displays tweets about a certain keyword in a table, and allows to play them if they contain a link to a YouTube video.

```
Find tweets about Lugano.
When an item is selected, extract youtube video from it, and
play video.
```

Listing 3.3. A player for videos linked from tweets.

The above recipe examples all conform to the CNL grammar and vocabulary and can be automatically executed by *NaturalMash*.

3.2.1 Grammar and Syntax

Figure 3.2 illustrates the grammar of the *NaturalMash* CNL represented in Extended Backus–Naur Form (EBNF). The top-level structure of a mashup recipe is decomposed into paragraphs, which are, in turn, a collection of sentences. The CNL accommodates specific grammatical constraints that allow only two types of sentences to be built:

- **Imperative sentences.** They are composed of multiple imperative mood clauses called *commands*. As a syntactic structure, commands should be separated by commas or “and”. For example, “Find tweets about Lugano, and find YouTube videos about Lugano” is an imperative sentence composed of two commands (separated by “and”).

⁷<https://dev.twitter.com/>

```

(*MASHUP DESCRIPTION*)
nlmd = paragraph (EMPTY_LINES paragraph)* (EMPTY_LINES? EOF);

(*PARAGRAPH*)
paragraph = sentence+;

(*SENTENCE*)
sentence = imperative | causal;

(*IMPERATIVE SENTENCE*)
imperative = command (command_delimiter command)* DOT;
command = (*natural language description of a Task*)
command_delimiter = COMMA AND?;

(*CAUSAL SENTENCE*)
causal = WHEN case COMMA imperative;
case = (*natural language description of an Event*)

```

Figure 3.2. The grammar of NaturalMash CNL represented in Extended Backus–Naur Form (EBNF).

- **Causal sentences.** They are written in causal form in which the time conjunction “when” introduces a passive clause (called a *case*) followed by one or more *commands*. The *case* describe the cause of the event (e.g., an event triggered by user interactions with a widget); the *commands* represent the effect to be realized when the *case* happens. Like in imperative sentences, *case* a *commands* should be separated by commas or “and”. As an example, consider the causal sentence in the Listing 3.1: “When an item is selected, search youtube videos about title”.

Commands and *cases* are built according to the vocabulary of the language that is derived from the *NaturalMash* component model. We will explain the component model in Section 3.3.

3.2.2 Semantics

Like its grammar, the CNL is compiled to an executable language with both *imperative* and *event-driven* semantics. Its imperative aspect is that the natural sequential order of the sentences in English (i.e., from left to right) defines the control flow of a mashup from one sentence, or one *command* to another.

The event-driven execution semantics of the *NaturalMash* CNL is modeled by causal sentences. Causal sentences are activated (but not immediately executed) when the main control flow reaches them. An activated causal sentence is ready to later receive control whenever its *case* happens, after which the control is immediately passed to its imperative part. In doing so, the imperative part initiates a parallel and independent control flow, which can be repeatedly executed for every occurrence of the Event. We will explain the compilation process of the CNL in detail in Chapter 4.

3.3 NaturalMash Component Meta-Model

Mashup ingredients (Web APIs) not only span all tiers (user interface, application logic, and data) of a Web information system [117], but within each tier they need to be composed using various techniques (e.g., Web scraping [102], Web clipping [104], synchronous remote/local service invocation, asynchronous interaction or subscription to feeds and data streams). Furthermore, different Web APIs can be made accessible through different access methods and technologies (e.g., POX, REST [99], or SOAP). As a consequence, one of the main challenges of mashup tools consists of abstracting away such heterogeneity behind a highly abstract, expressive, and extensible component meta-model. A highly abstract component meta-model conceals as much as possible the information about the technical details of its underlying ingredient. This abstraction, however, should not compromise the expressiveness of the meta-model, which is defined as how many different types of ingredients can be defined by the meta-model. Moreover, new technologies and access methods emerge frequently on the Web (such as WebSockets), hence, it is important that the meta-model is easily extensible. These challenges are not fully addressed by the state-of-the-art meta-modeling solutions for mashup ingredients.

The Enterprise Mashup Markup Language⁸ (EMML), for instance, is a XML-based standard component meta-model for mashup components providing services and data sources. Though it supports variety of service and data source types, it does not support ingredients with a graphical interface (widgets) so lacks a high level of expressiveness. JOpera [94], on the other hand, is based on a highly expressive meta-model called Opera Modeling Language (OML). For modeling widgets, however, OML is not abstract enough. The meta-model presented in [95] is able to describe not only different types of services (HTTP/REST

⁸<http://www.openmashup.org/>

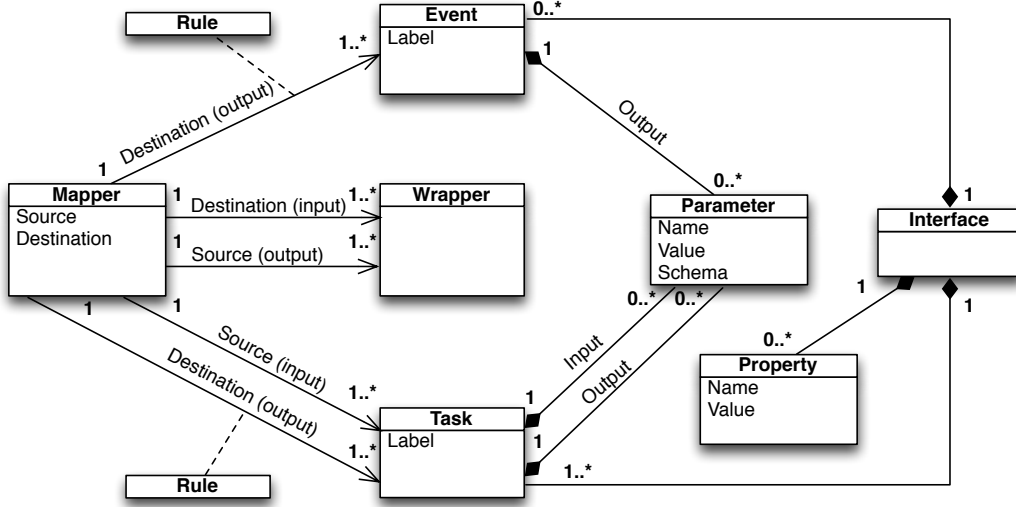


Figure 3.3. A meta-model for ingredients (Web APIs).

and SOAP) but also widgets at an acceptable level of abstraction. One disadvantage of the work is nevertheless the lack of support for extensibility.

In this section, we present our component meta-model that is highly abstract, expressive, and extensible. To be specific, our component meta-model is responsible for: (i) modeling heterogeneous ingredient technologies in a unified and extensible way, and (ii) describes the capabilities of ingredients in natural language to form the vocabulary of the CNL.

3.3.1 Modeling Ingredient Technologies

Figure 3.3 depicts our component meta-model as a UML class diagram, whose main classes as well as the containment and reference relationships existing between them will be described in the rest of this section. The six classes of the meta-model (interface, task, event, parameter, mapper, and wrapper) form two layers of abstraction on top of a given ingredient. The first and underlying layer is provided by the wrapper class which models various invocation and composition styles. The second layer is achieved through the interface class which contains the event and task classes, which, in turn, contain the parameter class. This layer provides a homogeneous, easy-to-understand interface concealing the underlying technological heterogeneity. The connection between these two layers is thus modeled with the mapper class.

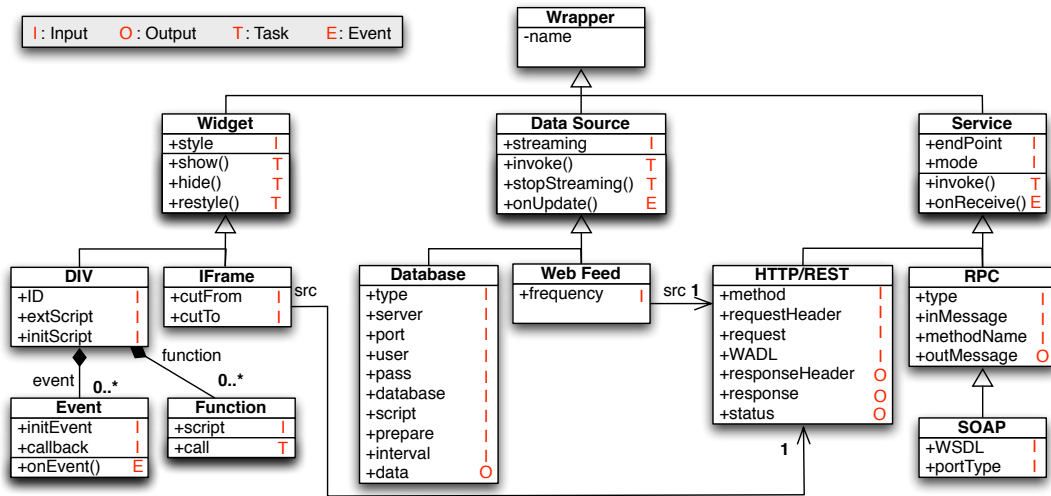


Figure 3.4. Wrapper hierarchy.

- **Interface** class enables interactions between ingredients. An interface may also contain some properties that identify the ingredient, exemplified by publisher, description, version, and call rate limit. As it will be mentioned in Chapter 4, the last properties is required to inform the *NaturalMash* compiler engine about any call rate limit imposed by the ingredient.

- **Task and Event** classes are attached to the interface of an ingredient and represent its passive and active operations. The task class models the functional operations of the ingredient (called a Task), and represents both the synchronous invocation of a functionality (such as a call to compute in some input) and a data source that generates some output given a set of input data. From the control flow point of view, a Task is considered passive inside a mashup, as it is executed only when the control reaches it. The task class may contain abstract *parameters* associated to its input and output ports. Input parameters should be filled with values before the invocation. The output parameters will eventually store the results of the invocation if it does not throw any exception.

The passive operations of ingredients called and Event, is modeled by the event class. Events fire independently, and can be used to represent updates from sources of streaming data as well as events caused by user interactions with a Web widget. Events may produce output data in their associated output port parameters, even if it is possible to model signal-like Events which do not carry data but are nevertheless used to trigger the execution of other Tasks.

The task and event classes have two similar attributes. The *name* attribute is a human understandable string. The *description* attribute holds the formal natural language description of the corresponding Task or Event. These formal descriptions that form the vocabulary of the CNL, are explained in Section 3.3.4.

- **Parameter** class can be marked as either input or output, and is bound to the task and event classes. The attributes of this class include: name that is a human-readable string, schema that is used to facilitate data integration (see Chapter 4, Section 3.4), and default value that is optional.

- **Wrapper** class is responsible for modeling different types of ingredients, distinguished by the technology for their invocation and utilization (Figure 3.4). Separating the wrapper class (technology specific part) from the interface class (technology independent concerns) raises the level of abstraction and also ensures extensibility in future by adding more wrapper types in an ad-hoc fashion. The wrapper classes are organized into a hierarchical inheritance structure. At the top of the hierarchy is the abstract wrapper class, which is inherited by three subclasses each of which corresponds to a different type of ingredients: *data source*, that delivers a snapshot or a stream of data from remote sources on the Web (e.g., BBC News), *service*, that provides remotely accessible business logic (e.g., Yahoo! Placemaker), and *widget*, that is a stand-alone Web application with a self-contained (reusable) user interface (e.g., Google Maps widget).

The three main wrappers (widget, service, and data source) have, in turn, a set of subclasses that are called *wrapper types*. Each of these types is associated with a single specific technology or access method. For instance, the database class, which is a wrapper type defined under *data source*, is used to model the interaction with a database. To fully model an ingredient may require more than one wrapper type depending on how many different technologies are required for invoking the target ingredient. We attempted to make the meta-model as expressive as possible by including in the wrapper class all the technologies and access methods utilized by existing mashup ingredients.

The wrapper types store specialized configuration data structures that model different interaction protocols of ingredients. Such data structures include a set of attributes that are conceptually grouped into *input*, and *output*, as well as a number of operations that operate on these attributes. Similar to the interface layer, these operations are interpreted as either Task or Event. Therefore, a modeled ingredient has two types of operations: one defined at the interface level, and the other at the wrapper level. Unlike the interface operations, the wrapper operations are hidden from end-users and are predefined according to the wrapper type. The same ingredient can thus be modeled differently at the interface level, whereas the wrapper part is always the same.

Wrapper Type: Widget

A Web widget consists of an independent visual presentation of its underlying aggregate data. Some widgets may also give a mashup access to reusable pieces of functionality or data. A widget life-cycle starts with calling the `show` Task, which initializes all input attributes with value, and ends when the `hide` Task is called. During this life-cycle, the `restyle` Task can be called multiple times to change the value of the `style` attribute containing a valid CSS stylesheet.

- **DIV widgets** are created on the client-side using external and internal JavaScript code. This kind of widgets is well exemplified by Google Maps APIs offering an external JavaScript library to build customized map visualizations. The `ID` attribute of the wrapper type specifies the actual ID of the DIV element that is supposed to contain the widget. The URL pointing to the external JavaScript libraries can be inserted as the value of the `extScript` attribute. The `InitScript` attribute needs to be set by the JavaScript code that actually wraps the widget in the DIV element.

The function class models JavaScript functions. This class has a Task operation (`call`) that executes the script specified in the `script` attribute. The operation can be then associated with an interface task whose parameters can be mapped and used inside the code as regular variables using the template rule (Section 3.3.2).

In addition to functions, a DIV wrapper can have a set of event objects representing events in JavaScript. In JavaScript, events are handled through the callback mechanism, in which a callback function is called whenever its corresponding event has occurred. The declaration and implementation of the callback function is the value of `callback` attribute. The script that attaches the callback function to the event should be then inserted as value of `initEvent` attribute.

- **IFrame widgets** are used to embed HTML-based user interface contents from a remote website, partially (i.e., Web clipping) or completely. The value of `src` attribute is a HTTP wrapper used to fetch the website (with `Get` method) that is to be embedded inside the target container. In case of Web clipping, the `CutFrom` attribute contains the starting point of the clipping process that can be a piece of string which matches a part of the target Website HTML code. Likewise, the `CutTo` attribute represents the ending point of the cutting process. For instance, cutting a table from a website in the simplest case requires setting `<table` and `>table>` as the values of the `cutFrom` and `cutTo` attributes.

Wrapper Type: Data Source

A data source provides remotely accessed content on the Web. Data coming from remote sources can also have a real-time essence. In this case, the wrapper allows the streaming of data from a data source by setting the `streaming` attribute to `true`. This should be done through the first call to the `invoke` Task which initiates all the input attributes and starts the ingredient life-cycle. If streaming is not enabled, the ingredient life-cycle ends as soon as the response from the `invoke` Task arrives. Otherwise, it will only end by calling the `stopStreaming` Task. Moreover, while streaming is activated, the `onUpdate` event is fired whenever a new data stream item is available.

- **Database** can be used to generate a stream of data. In the simplest case (which is the only case supported by our meta-model), this can be done by running a query (script) in a given time interval. This interval is specified by the `interval` attribute, and its length is a factor of how fast the target data changes.

- **Web Feeds** is a popular type of ingredient for building data mashups, and is usually delivered in RSS/Atom format. Feeds can be consumed as a continuous stream of data using a technique called polling. Using this technique, the feeds are periodically checked for updates. The feed wrapper type uses the `frequency` attribute to specify the update interval.

Wrapper Type: Service

Services are reusable modular business logic that are made available to remote clients through a URL (`endPoint`). The interaction between a client and a service provider (mode) can be either blocking, in which the invocation of the service (by calling `invoke()`) must wait until the response is ready before it can proceed, or non-blocking, in which the client invokes the service and registers a callback (`onReceive()`) to be triggered whenever the response comes back.

- **RPC/SOAP** has three popular forms: SOAP, XML-RPC⁹, and JSON-RPC¹⁰. All of these protocols run over the HTTP or secure HTTP (HTTPS), except for SOAP messages that can be also transmitted over SMTP and other protocols. The protocol in use, therefore, can be found at the URL to the service endpoint (`endPoint`). To communicate requests (`inMessage`) and responses (`outMessage`), JSON-RPC messages use JSON, whereas SOAP and XML-RPC messages are encoded as XML. To determine the type of the RPC service, the

⁹<http://www.xmlrpc.com/>

¹⁰<http://json-rpc.org/>

type attribute can be set to “SOAP”, “XML-RPC”, or “JSON-RPC”. In case of SOAP, a service usually comes with a Web Services Description Language (WSDL) document (WSDL attribute), which defines the service interface. In case that the interface contains more than one port types, `PortType` specifies the desired one.

- **HTTP/REST** is the communication protocol used on the Web. Following the constraints of the REST architectural style, it is used for stateless client-server interaction with Web servers by sending and receiving hypermedia documents (e.g., XML, HTML, etc.) via four methods (GET, POST, PUT, and DELETE). The HTTP communication mechanism is based on request and response messages, each of which is decomposed into a header (`requestHeader` and `responseHeader`) and a body (`request` and `response`).

3.3.2 Mapper and Rule

The *mapper* class serves as a mechanism providing a link between the predefined wrapper operations and the interface operations. This is made possible by defining how the wrapper attributes and the interface parameters are mapped to each other. This class contains two attributes: `destination`, and `source`, which hold a reference to parameters and wrapper attributes respectively or vice versa, depending on the mapping direction.

Using the mapper class, abstract parameters can be passed from an interface as input to a wrapper Task operation which controls the technology-dependent invocation of its corresponding ingredients. Likewise, the attributes holding the result of the invocation of a Task, or the data associated with an Event operation, are transformed to the interface as abstract parameters. The *rule* class is used to designate how the data mapping from a source to its destination should be accomplished. This class contains three subclasses each of which represents a specific rule prescribed for a particular purpose:

- **Correspond** rule enables a one-to-one association between a source and its destination. In doing so, the value of the source is exactly copied to its destination parameter. This requires both source and destination have the same data type, or otherwise be implicitly convertible. For instance, the numerical type or the majority of Media types (such as XML, JSON, etc.) can be converted to the string type. For example, `User` and `Pass` attributes of database wrapper can be set at run-time by an abstract Task taking two input parameters of string type, which are mapped with a one-to-one correspondence to these attributes.

- **Template** rule can be selected to correspond multiple sources to single destination through a template. This template string consists of the value which is supposed to be assigned to the destination parameter, as well as a set of place-

```
URL = http://search.twitter.com/search.atom?q={query}
```

Listing 3.4. Template rule applied to map the interface parameter “query” (source) to the “URL” attribute of the HTTP wrapper (destination).

holders distributed within the string value. These placeholders are represented as pairs of opening and closing brackets, each of which contains the name of a single source parameter or attribute which will be replaced with its value at runtime (i.e., during the execution of the ingredient). One frequent use of the template rule is for passing URL parameters (when the corresponding WADL document is not present). Consider invoking the Twitter search API with the HTTP wrapper. Using the template rule we can pass a source parameter to the URL attribute of the HTTP wrapper (Listing 3.4).

- **Parse** rule entails splitting a source to multiple values and further associating these values to destinations. This rule supports different query languages. A query written in a specified language, is used for extracting the value of the destination from its source. The language can be one of XPath, exclusively for XML-based formats (e.g., SOAP, RSS, ATOM, XML-RPC, etc.), or a regular expression for scraping data out of any kind of text-based formats such as HTML or JSON. This rule is specially useful for extracting data from Media types such as SOAP messages, RSS/Atom, and XML.

3.3.3 JSON Schema for The Component Model

Listing 3.5 is the JSON schema corresponding to the component meta-model depicted in Figure 3.3. Listing 3.6 is an example of a component model in JSON for the Twitter API. We choose to rely on the JSON syntax not only to take advantage of many related tools and libraries but also because ingredient libraries will be mainly managed from within mashup composition tools running inside a Web browser, where JSON¹¹ is one of the most efficient data representation and interchange formats.

The schema contains the classes of the meta-model. The abstract interface class is represented as interface object containing an array for defining its properties, as well as two arrays: one for the Tasks and another for the Events. Parameters are also defined as items of an array which is named as output and can be included in every Task or Event item. The Task items can also contain another array of parameters called input, since Tasks in the meta-model, as

¹¹<http://www.json.org/>

```

{ Interface: {
  name: "",
  property: [{name: "", value: ""}],

  task: [{name: "", description:"",
    input: [{name: "", value: "", schema: ""}],
    output: [{name: "", value: "", schema: ""}]
    }],

  event: [{name: "", description:"",
    output: [{name: "", value: "", schema: ""}]
    }]
},

service_wrapper: [{type:"", name:"",
  input: [{name: "", value: ""}],
  output: [{name: "", value: ""}]
}],

datasource_wrapper: [{type:"", name:"",
  input: [{name: "", value: ""}],
  output: [{name: "", value: ""}]
}],

widget_wrapper: [{type:"", name:"",
  input: [{name: "", value: ""}],
  output: [{name: "", value: ""}]
}],

mapper: [{source : ["ref"],
  destination: [{name:"ref",
    correspond:"ref" | template:"" |
    parser:{ language:"", query:""}}]
  }]
}

```

Listing 3.5. JSON schema for the component model.

opposed to Events, can also take input parameters. The parameter items in the JSON schema take name, value, and type, similar to the abstract parameter class in the meta-model.

The arrays `service_wrapper`, `datasource_wrapper` and `widget_wrapper` express, respectively, service, data source, and widget wrappers in the meta-model. Each array can have multiple items referring to different wrapper types

defined under their meta-model class. These items are identified by the type variable that takes the wrapper type name as a string such as “HTTP”, “DIV” and “SOAP”. The input attributes of a wrapper type are defined in the input array, only when they should be initialized by a constant value.

Finally, the mapper class is expressed as an array whose items identify mapper objects. The variables `source` and `destination` make a reference to either a parameter name or wrapper attribute. To ensure their uniqueness, references can be prefixed with the names of the object they belong to. This way they can also be referenced from the mapper objects. To exemplify this, `user` which is an input attribute of the database wrapper can be referred to by `dbwrapper.user`, assuming `dbwrapper` is the name of the wrapper object. Likewise, references to parameters can be prefixed by the name of the Task/Event they belong to.

3.3.4 Formal Description of ingredients in Natural Language

The *label* attribute of each interface operation is associated with a formal textual description (called a label) that not only explains the capabilities of the ingredient in a human readable text format but also forms the vocabulary of the CNL. Labels are defined differently for Task and Events. In case of Events, the label should be a subject-verb-object clause like “an item is selected”, a label associated with the item-selection event of a table widget. As for the vocabulary, a *case* is exactly an Event label.

For Tasks, a label should be an imperative clause. For instance, “find slides about [keyword]” is a label assigned to the search-slide Task of the Slideshare API. Moreover, the name of each input parameter (keyword in this case) should be enclosed within square brackets, creating a placeholder for an object. In a mashup recipe, an object to fill a placeholder can be a parameter name referring to the output of previous Tasks, a constant value (e.g., a string), or an anaphora — in linguistic, an anaphora (e.g., “that”) is defined as an expression linking two elements in a document — pointing to a specific part of the mashup description text (e.g., in Listing 3.3 the clause “extract video items from it” contains the anaphora “it” pointing to the output parameter of the event produced by the table Widget). A *case* is built from a Task label by replacing the placeholders of the label with objects. For example, given the label “find songs titled [keyword]” the corresponding clause can be “find songs titled mashup”, where the object is replaced with a constant value (“mashup”).

```

{ Interface: {
  name: "Twitter",
  property: [{name: "description", value: "It allows to access core
    Twitter data"},
    {name: "publisher", value: "Twitter"},
    {name: "version", value: "1.1"},
    {name: "call-rate-limit", value: "150"},
    {name: "call-rate-limit-metric", value: "hour"}],

  task: [{name: "searchTweets", description: "find tweets about {keyword
    }",
    input: [{name: "keyword", schema: ""}],
    output: [{name: "tweets", schema: ""}]
    }],

  service_wrapper: [{type: "http", name: "searchAPI",
    input: [{name: "method", value: "get"}],
    }],

  mapper: [{source : ["searchTweets.keyword"],
    destination : [{name: "searchAPI.url",
      template: "http://search.twitter.com/
        search.atom?q={query}"}]
    },
    {source : ["searchAPI.body"],
      destination : [{correspond: "searchTweets.tweets"}]
    }
  ]
}]

```

Listing 3.6. An example component model in JSON for the Twitter API. The schema attributes are intentionally left empty, as they will be thoroughly discussed later in Chapter 4, Section 3.4.

3.4 Ontology-based Data Integration Framework

Mashup ingredients are highly heterogeneous in terms of not only access protocol technology (discussed in the previous section) but also data format. They may receive and produce data in a variety of formats ranging from Media types to custom data formats. This poses a great challenge when integrating these heterogeneous data. Data integration is an important part of mashup development [11] and consists of converting and transforming data from one ingredient to another. Therefore given this data format heterogeneity, manually integrating

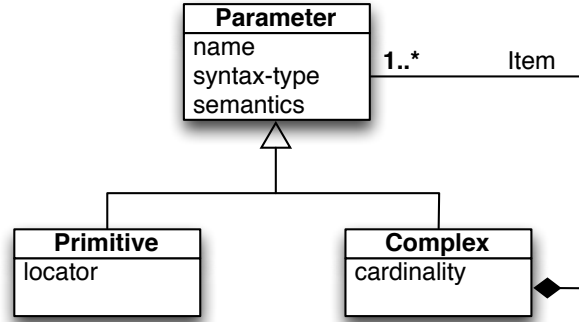


Figure 3.5. The meta-model to which parameter schemas conform.

these data is cumbersome and error-prone. Also, a study [88] shows that non-programmers can hardly grasp the concept of data flow. Accordingly, mashup tools should be able to automate data integration as much as possible.

The challenge of facilitating data integration has been addressed differently before [37]. In most mashup tools data integration is done semi-automatically. Semi-automatic data integration means that the frameworks provide some recommendations and the end-user has to intervene and pick one. Yahoo! Pipes, for example, provides some data flow hints according to the syntax (format) of the data. Recent tools like DashMash [25] also consider the semantics of the data when providing data flow hints.

We propose a new semi-automatic approach based on both syntactic and semantic aspects of data. We emphasize the openness of the approach which has received little attention so far by similar previous works. Openness ensures the extensibility of the approach in terms of adding new data formats and schemas. In the rest of this section, we will describe our approach that is implemented in a framework integrated in *NaturalMash*. The framework consists of a specification language for schemas, formalities to define new data formats, and a mechanism to provide data flow hints.

3.4.1 Schema Specification and User-defined Data Formats

In *NaturalMash*, data are communicated through input/output parameters attached to ingredients. The core of the framework lies in associating schema to these parameters that facilitates their integration. A parameter schema is a structure of metadata determining how to access and interpret the value of the parameter.

```

{
  complex: [{
    name: ""
    syntax-type: "",
    semantics: "",
    cardinality: "",
    items: [{schema-ref: ""}]
  }],

  primitive: [{
    name: ""
    syntax-type: "",
    semantics: "",
    locator: ""
  }]
}

```

Listing 3.7. Parameter schema meta-model represented in JSON syntax.

Figure ?? shows the meta-model to which a parameter schema is conformed. The schema attributes in a component model will be populated by parameter schemas. Therefore, in order to be consistent with the component meta-model (Section 3.3.3), we also use JSON syntax (Listing 3.7) as the meta language to specify parameter schemas. For instance, Listing 3.8 is a data schema associated with the output parameter (tweets) of the Twitter API model (Listing 3.6).

According to our schema meta-model, parameters are categorized into either primitive or complex. Primitive parameters are similar to standard Java variables with a primitive data type (e.g., integer, double, and float). The “locator” attribute of a primitive parameter is optional and contains a regular expression to extract the value of the parameter. The name of the schema (“name” attribute) should match the name of the parameter. In the JSON representation of a primitive schema, only the “primitive” array exists with one element associated with the main schema.

The structure of a complex parameters is a data table composed of more than one primitive or complex parameter called items. In the JSON representation, the “complex” array should have at least one element (main sub-schema) corresponding to the parameter (the “name” attribute should be the name of the parameter), and one or more sub-schemas of either primitive or complex type. The sub-schemas are referred to in the “items” array by their names. For instance, in the Twitter example (Listing 3.8) the “items” array of the main sub-schema contains references to the “tweet” and “user” sub-schemas of primitive

type. If an item makes a reference to a primitive sub-schema, the “locator” attribute becomes mandatory and defines the path (separated with dots) to the item in the complex parameter (similar to XPath). Nesting can happen when an item refers to a complex sub-schema, in which case sub-schemas can be reused (two different items refer to the same sub-schema). The cardinality of a complex parameter (“cardinality” attribute) refers to the the number of times the data table is repeated inside it. For example the cardinality of RSS feeds as as complex parameter is “1..*”.

Syntax types (“syntax-type” attribute) are equivalent to data formats and can be primitive, user-defined, or Media types. Primitive data types (e.g., integer) are associated with primitive parameters or sub-schemas. User-defined and Media types are only assigned to complex parameters or sub-schemas. The framework supports adding new user-defined types in an ad-hoc manner, following the syntax shown in Listing 3.9. The items of a user-defined data format can be either optional or mandatory (by setting the “optional” attribute to, respectively, true or false). Unlike an optional item, a mandatory item should be present in data with the user-defined format For example, Listing 3.10 defines a data format to store a geographical location. It contains two mandatory (“longitude”, and “latitude”) and two optional items (“radius”, and “address”).

finally, the framework allows to define the semantics (“semantics” attribute) of parameters using the ontology proposed by [47]. In doing so, we also promote the practice of reusing existing ontologies. However, we also allow to modify the ontology in order to make it domain-specific (the target application domain such as bioinformatics, e-learning, and health care), thus improving the accuracy of data integration. In the schema, the semantics of a parameter (or an item) are defined by a set of tags consisting of concept labels from the ontology. For example, the semantics of the “tweets” parameter is defined by the concept label “Text” which, in the ontology, is a subclass of “Media_Content” concept.

3.4.2 Integration Process

The framework facilitates data integration by generating a shortlist of candidate output parameters to feed a given input parameter. In case of more than one candidate, the end-user has to manually select one of them (disambiguation). The process of data integration is broken down into two steps. The first step is called *schema mapping* in which one or more output parameter schema (source schema) are mapped to an input parameter schema (target schema). The mapping can be one-to-one (one source schema to one target schema), or many-to-one (more than one source schema to one target schema). All the combinations

```

{
  complex: [{
    name: "tweets"
    syntax-type: "application/json",
    semantics: "Text",
    cardinality: "*",
    items: ["tweet", "user"]
  }],

  primitive: [{
    name: "tweet"
    syntax-type: "string",
    semantics: "Message",
    locator: "text"
  },
  {
    name: "user"
    syntax-type: "string",
    semantics: "People",
    locator: "user.name"
  }]
}

```

Listing 3.8. An example of a parameter schema for the Twitter API.

of source and target parameter types can be mapped as follows:

Primitive to primitive is a one-to-one mapping. All the data types can be converted to each other automatically. Only in case of double (source) to integer (target) conversion, the fraction part is omitted.

Complex to complex mapping is performed in a one-to-one fashion. We consider all markup Media types (e.g., XML, JSON, and HTML) convertible to each other. When mapping a repetitive output parameter to a non-repetitive input parameter, only the first repeated element of the output parameter is considered for matching.

Complex to primitive mapping is one-to-one. In this mapping, only one item of the output complex parameter can be matched with the input primitive parameter.

Primitive to complex mapping can be either one-to-one or many-to-one. In doing this mapping, the output primitive parameter is matched with one of the items of the input complex parameter.

```
format: {
  name: "",
  cardinality: "",

  items: [{
    name: "",
    syntax-type: "",
    optional: true | false
  }]
}
```

Listing 3.9. The syntax to define new data formats.

The second and final step is *schema matching* that consists of one-to-one association of items (or primitive parameter) of two mapped schemas. The matching is achieved at the following levels:

Syntactic level in which the source and target schema have the same or convertible data types.

Semantic level in which the defined semantic tags (the ontology concepts) allow to match two schemas at this level. The framework uses the Peller semantic reasoner [103] on the source and target schemas to compute any subsumption relation between them.

Name level in which two schemas are matched at this level, if their names are similar. The framework uses the SimPack [21] library to compute the similarity between the names.

A matching between two items or primitive parameters is achieved when they match at least at one of the levels above. The matching score for a schema mapping is calculated by counting the number of obtained matching levels. The mappings for each input parameter will be sorted by their matching score.

3.5 NaturalMash Composition Environment

The *NaturalMash* environment is designed to provide an innovative selection of features that are meant to enhance the user experience and the ability of end-users to build sophisticated mashups. The design of the environment has been evolved over two years, as a result of a formative user-centered process. In this section, we consider the current version of the environment as this dissertation


```
format: {  
  name: "location",  
  cardinality: "1",  
  
  items: [{  
    name: "longitude",  
    syntax-type: "float",  
    optional: false  
  }, {  
    name: "latitude",  
    syntax-type: "float",  
    optional: false  
  }, {  
    name: "radius",  
    syntax-type: "float",  
    optional: true  
  }, {  
    name: "address",  
    syntax-type: "string",  
    optional: true  
  }  
]  
}
```

Listing 3.10. An example of a user-defined data format storing a geographical location.

is being written. We postpone the details of the evolution and evaluation of formative environment to Chapter 5. In the following, we first describe each feature individually and later show in a usage scenario how they are used in conjunction to build a mashup.

- **Inline Search** feature facilitates ingredient discovery directly in the text field (Figure 3.6) and allows end-users to (i) type in the text editor the (approximate) name of the ingredient they are looking for, which results in the end-user getting a list of labels associated with the ingredient matching or approximating the given keyword, or (ii) type what the ingredient is supposed to do (in case they do not know or cannot guess the exact name of the ingredient), by doing which the input text will be matched against all the labels associated with all the ingredients in the library. In the latter case, the mechanism of searching labels is based on exact match, word synonym (e.g., “search” and “find”), or word semantics (e.g., “location” and “map”).

- **Autocompletion** feature lifts the learning barriers of the CNL caused by its grammatical and vocabulary (labels and objects) constraints (Figure 3.6). Based

on what end-users type in the text field, a list automatically appears and shows suggestions for Task/Event labels (to support ingredient discovery and reuse), and data flow (i.e., referencing suitable objects within Task labels).

[tweet].

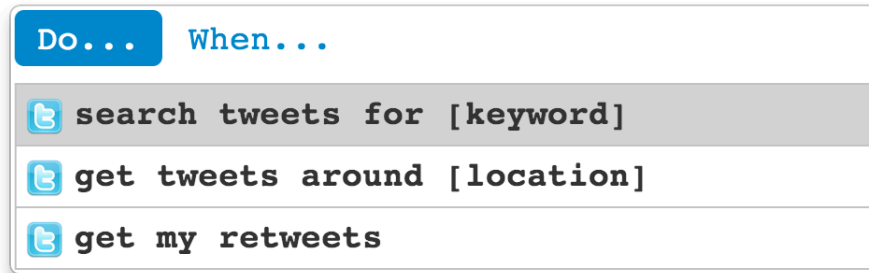


Figure 3.6. Typing “tweet” results in the autocomplete list showing the labels associated with the Twitter API.

- **Semi-structured Text Field** feature supports the end-users’ learning experience and ensures that end-users’ input will not cause syntax errors, while still allowing a high degree of freestyle editing. To be specific, the text field: (i) restricts input characters to avoid accidental syntax errors (for instance the new line characters are disabled while typing an object in a placeholder), (ii) automatically inserts the separators (“,”, “and”, and “and,”) if the cursor is positioned before and after clauses (manual insertion of the separators is also possible), and (iii) streamlines selecting and moving text objects (clauses) via, respectively, double-click and drag-and-drop.

- **Data Flow Highlighting** feature makes the objects indicating flow of data display in boldface (Figure 3.7). Moving the cursor on an object results in the highlighting of the object source Task or Event. This way, users can discover the source of an object both when browsing the data flow suggestions in the autocomplete list but also after a data flow suggestion has been inserted.

- **Error Highlighting** feature, similar to many “spell-checking text editors”, shows an error using a red wavy line under the text that produced it. For example, if there is an ambiguity in the input text (e.g., the input does not match any Task or Event label), the compiler produces an error that is reported to the end-user as the text is being entered (Figure 3.6). Whenever the end-user moves the cursor (or click) on the highlighted erroneous text, an autocomplete list containing possible suggestions to disambiguate the label is shown, thus offering the opportunity to the end-user to quickly correct the mistake.

`when the map is clicked` , get tweets around `location` .

Figure 3.7. The source of the object (the “map click” label) as well as the object itself (“location”) get highlighted as soon as the cursor is placed in the object text.

- **Drag-and-Drop** feature allows end-users to drag-and-drop an ingredient from the toolbar into the text field, visual field, or ingredient dock. A drag-and-drop on an ingredient causes the autocomplete list to appear in the text field displaying the labels of the ingredient. If the ingredient is a widget, it will also be added and shown in the visual field.

- **Programming by Demonstration** feature allows to append an Event label to the mashup recipe by just manually triggering it in the visual field. For instance, clicking on Google Maps widgets results in adding the text “when the map is clicked” to the recipe. To grab the attention of end-users, the label is highlighted both after it has been added and every time its Event is triggered.

- **Synchronized Multi-perspective Modeling** feature keeps the three main interaction components of the environment (ingredient dock, text field, and visual field) synchronized during every user interaction: (i) editing text in the field updates the visual field and the ingredient dock; (ii) selecting a widget from the visual field or a ingredient from the ingredient dock results in highlighting its corresponding text in the text field and vice versa (moving the caret through a portion of the text highlights its associated widgets and ingredient icons); (iii) deleting an icon from the dock or a widget from the visual field results in the removal of its corresponding text (and vice versa).

3.5.1 Usage Scenario

The following illustrates a common and complete usage scenario of *NaturalMash*, whereby an end-user builds the mashup example described in Listing 3.1.

The first step is to discover the right ingredients for finding slides. This step can be facilitated by the inline search feature which enables the end-users to type what the ingredient he is looking for is supposed to do. For instance, the end-user can start by typing “search slides”, which results in the text field providing an autocomplete list of labels that contain the input words or synonyms for the words. Once the autocomplete list is displayed, the end-user can select a proper suggestion (in this case, “find slides about [keyword]”) by either pressing the Enter key or by pointing with the mouse and clicking.

After selecting a label from the autocomplete list, (i) the label is inserted in the text field, (ii) ambiguity is resolved, in case there are one or more similar labels, (iii) the mashup is rebuilt and executed, (iv) another autocomplete list containing data flow suggestions for the label is displayed. For the input parameter (“[keyword]”), the end-user may type a constant string like “APIDays” resulting in a mashup that uses Slideshare API to search for slides and document matching the input constant, and automatically shows the results in the Slideshare widget.

The output mashup is interactive and supports PbD in a way that, for instance, clicking an item in the Slideshare widget results in not only showing the item in the embedded frame of the widget, but also appending the corresponding Event label (i.e., “when an item is selected”) to the text field as well as setting the focus in a way that makes it easier for the end-user to add some Task labels to complete the causal sentence. For example, the end-user may type “video” in the text field or, alternatively, search for the YouTube API in the ingredients toolbar and then drag-and-drop the ingredient to the text field, both of which result in displaying an autocomplete list containing the YouTube API labels. Immediately after selecting a suggestion, another autocomplete list containing data flow suggestions is shown. The end-user can select the output parameter “slide title” from this list, or type an anaphora pointing to the item such as “it”, both referencing the click Event label of the Slideshare widget. The data flow highlighting feature helps end-users to figure out the source of an object. Leaving the object placeholder empty results in a compiler error that is shown by a red wavy line under the placeholder. Clicking on the wavy red lines displays the autocomplete list associated with the placeholder.

While typing the mashup recipe, the end-user may modify the mashup user interface layout in the visual field. The final mashup can then be deployed in production, with a single click. Even after a mashup has been published, it can still be modified and redeployed at any time.

Chapter 4

The Architecture of NaturalMash

In this chapter, we explain the architecture and implementation of *NaturalMash*, which rely on model-driven engineering [69] and natural language processing techniques and libraries [42]. We first outline a set of challenges that confront the design feasibility of the system (Section 4.1), including minimizing the overall response time, and then comprehensively explain our proposed architecture to address these challenges (Section 4.2). Our client/server architecture is primarily based on caching and incremental compilation.

4.1 Design Challenges for Live Mashup Tools

NaturalMash is designed as a live mashup tool [7], which completely automates the repetitive task of compiling, deploying, and running mashup recipes at the level 4 of liveness (see Chapter 2). Despite the benefits of live mashup tools [7], their design and development confront two major challenges that we seek to address in this chapter:

Minimizing the live response time. Response time is the time elapsing between the instant the end-user manipulates a mashup being created to the instant the resulting mashup is compiled, executed, and displayed to the end-user. End-users may be detecting latency in the execution response of a live mashup tool because of a variety of factors. Some of these factors are concerned with the real performance of the system. It means how fast the system can compile and execute the mashup being created. There are also factors that deal with the way the live execution response time information is managed and provided at the end-user interface level. These refers to the perceived performance of the system. Failing to minimize the live

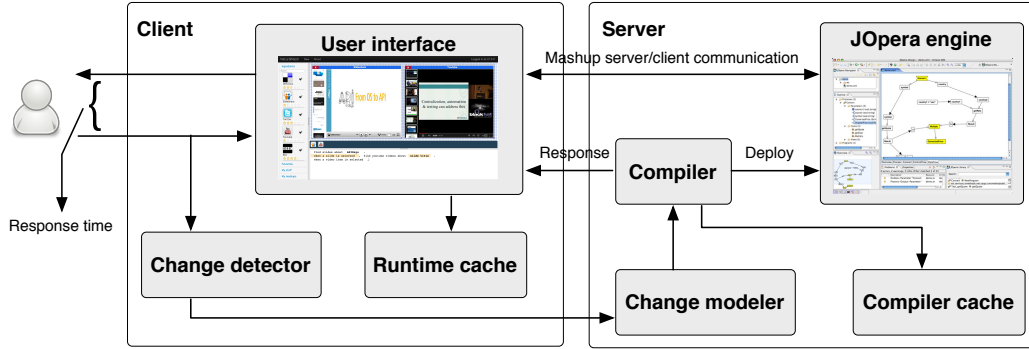


Figure 4.1. The liveness-friendly architecture of NaturalMash aims at supporting live programming by minimizing the response time and providing a mechanism to cope with the service call rate limits.

response time through increasing both real and perceived performance of live mashup tools may cause anxiety in their end-users.

Coping with service call rate limits. The majority of free mashup ingredients introduce strict call rate limits. For instance, Twitter API, which is one of the most popular mashup ingredients, allows 150 calls per hour. Moreover, many existing ingredients still issues developer keys to identify the source of calls. Within a mashup tool platform (e.g., *NaturalMash*), an ingredient is usually developed and shared by one person (there is one developer key), whereas it might be used by many end-users at the same time. These issues may pose serious challenges on the feasibility of building mashups in a live fashion. This is because the live mashup development process requires continuous execution of the same mashup, and thus there is a risk of exceeding the call rate limit of the constituent ingredients.

4.2 A liveness-friendly Architecture

To address the above challenges, we present a client/server architecture for *NaturalMash* (Figure 4.1), with the goal of minimizing the response time and providing a mechanism to cope with the service call rate limits. The client-side runs in a Web browser and presents the user interface of the mashup tool and of the resulting mashup (in the visual field). The server-side handles the compilation of the recipes into executable representations and supports their runtime.

4.2.1 User Interface

The end-users interact with the system and experience the response time through the *user interface* of the system. The life-cycle of each interaction begins with the end-user making a single modification to the model of the mashup being developed on the client. This modification targets either the visual field (modifying the user interface) or the textual field (editing the recipe). Each modification may result in the server (re)compiling, (re)deploying, (re)executing, and (re)rendering the target mashup. While the end-users are waiting for a response, it is essential that the system user interface prevents any perceived performance anxiety experiences. To this end, we adopted the following guidelines:

- Enabling waiting control mechanism (e.g., showing a spinning wheel) after a reasonable time (e.g., after 150 milliseconds [32]).
- Showing the mashup user interface after the complete load.
- Not preventing end-users from interacting with the mashup while it is being compiled.
- Making end-users aware of Internet connection errors.

4.2.2 Incremental Change Detector of Mashup Models

Mashup compilation and deployment can be time-consuming tasks. Therefore, initially, and as soon as the end-user makes a modification to the mashup (e.g., editing the recipe), the *change detector* component (client-side) identifies whether or not the modification requires issuing a (re)compilation request. The mechanism behind this component is based on classifying possible modifications as follows:

- Front-end modifications are applied to the mashup user interface (e.g., reorganizing widgets within the mashup user interface layout, and adding or removing widgets), and therefore, can be handled on the client-side without a need for recompiling and redeploying the whole mashup. These modifications are temporarily stored and previewed, and once the user's session is finished or idle (to avoid losing the modifications in case of disconnection), the modifications are sent to the server for persistent storage.
- Incomplete modifications require further modifications to take a visible effect. These may leave the mashup recipe in a temporarily incorrect state

as the end-user needs to complete them before they can be executed. For instance, a new Task is being added, but since no data flow source has been bound to its input it is not yet possible to display its results in a suitable Widget. These modifications may trigger the display of the autocomplete list or the highlighting of errors so only a partial compilation is required.

- Logic modifications change the back-end of the mashup, and thus require to recompile, redeploy and re-execute the mashup on the server-side.

Once the change detector decides a recompilation is required, a request is sent to the server (over WebSockets). To facilitate and speed up the compilation process, the architecture includes the *change modeler* component that supports the notion of incremental compilation [20]. More in detail, a source model and a target model are required to generate a change model. The source and target models are, respectively, the high-level (abstract) and low-level (code) models of the mashup. The change model conforms to a meta-model that defines possible changes that may occur on the mashup (i.e., both on the source and target models) and contains information to link these changes from the source model to the target model. Since in many cases mashups are grown incrementally by adding one API at a time, it is possible to extend the low-level code without having to regenerate it from scratch.

4.2.3 Compilation and Deployment

The resulting change model is passed to the *compiler* component, which is responsible to (i) generate executable code corresponding to the change model, (ii) to merge it with the existing code, and finally (iii) redeploy the mashup. This component should allow to terminate an unfinished compilation process to avoid continuous compilation requests in short-intervals that puts a heavy load on the server.

The compiler component in *NaturalMash* provides a pipeline that transforms mashup recipes into executable models of Web service compositions that are executed by the JOpera engine [94]. In the following we briefly describe the main steps of the *NaturalMash* compilation process.

The process relies on a representation that initially contains the input recipe text, but later is augmented with a list of ingredients used by the mashup, the specifications of the layout of the mashup user interface (e.g., the size and position of the widgets), and an abstract syntax tree containing data flow information (i.e., source and destination of objects) and a mapping between the text

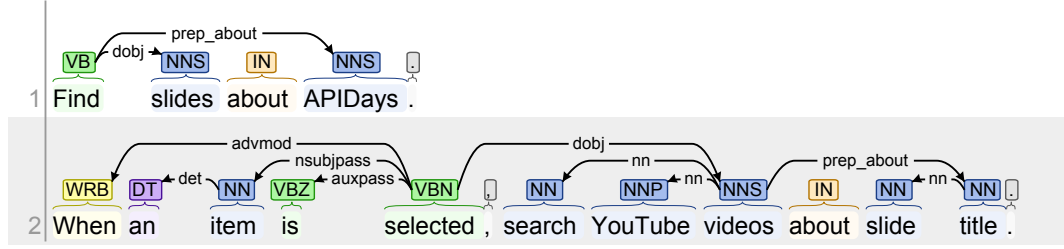


Figure 4.2. The visualized output of Step 1 (linguistic information) for the Listing 3.1 example using the Stanford CoreNLP online tool (<http://nlp.stanford.edu:8080/corenlp/>). The input is split into two sentences. Part-of-speech tags (VB: base form verb, NNS: noun plural, IN: preposition, DT: determiner, WRB: wh-adverb, NN: noun, VBZ: present verb, VBN: past participle verb) are associated with each word in the input text. Grammatical dependencies (det: determiner, advmod: adverbial modifier, nsubjpass: passive nominal subject, auxpass: passive auxiliary, dobj: direct object, nn: noun compound modifier, prep_about: prepositional modifier) are shown using arrows.

chunks (i.e., clauses and phrases) and Task and Event labels. The mashup representation is recycled with each round of compilation and is continuously updated as the mashup is being developed.

- **Step 1: Natural Language Parsing** The input recipe text is parsed and its linguistic information is extracted (Figure 4.2). This step is implemented using the Stanford CoreNLP library (<http://nlp.stanford.edu/software/corenlp.shtml>), which enables to tokenize the input text and split it into sentences, parse the text and assign a part-of-speech tag (verb, noun, etc.) to each word, process grammatical dependencies, and create the anaphora resolution graph.

- **Step 2: Constrained Natural Language Parsing** An abstract syntax tree based on the *NaturalMash* CNL grammar is produced. In this step, we use a formal lexer and parser (implemented using ANTLR, <http://www.antlr.org/>) to extract and identify sentence types as well as to extract their chunks (i.e., imperative or passive clauses).

- **Step 3: API Binding** The output of Steps 1 and 2 is consumed to build a mapping between the text chunks extracted in Step 2 and their corresponding Event/Task label. To attain this mapping, we first gather all the labels associated with the Tasks and Events of the APIs registered within the *NaturalMash* library. These are matched against the text chunks by ignoring the parameter placeholders. The result is a mapping between each text chunk and the corresponding Task or Event. In this step, ambiguity may occur when more than one

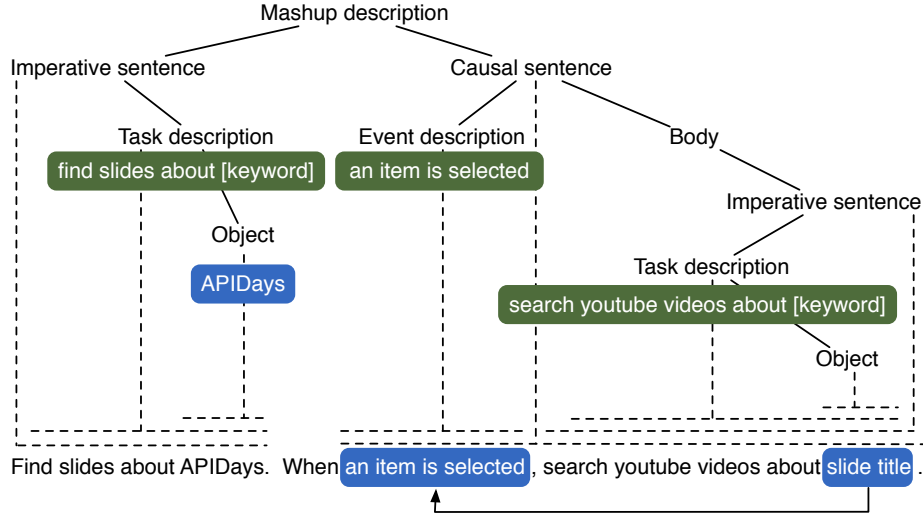


Figure 4.3. The annotated syntax tree corresponding to the mashup label of Listing 3.1.

Task/Event label match the same text chunk. Assuming that multiple APIs sharing the same label are equivalent, the ambiguity can be resolved automatically based on well known QoS-driven dynamic binding techniques [106]. Manual intervention through the system’s autocompletion feature is required only if there is an aliasing problem.

- **Step 4: Data Flow Resolution and Suggestion** The mapping generated from Step 3 is used to extract objects references and complete the syntax tree (Figure 4.3). To do so, the placeholders found within the Task labels representing input data are bound to the output data referenced from the actual text. Using the results of the linguistic analysis (Step 1) also the anaphoric objects are resolved. The data flow operations (e.g., matching and conversion) are delegated to the *NaturalMash* ontology-based framework for data integration that was explained in Section 3.4. At this stage, for each schema mapping, the framework dynamically generates the required code to carry out the data integration at runtime. This is done in a model-driven fashion using JOpera. We have implemented a set of JOpera adapters (extensions) that given some inputs, can perform a data integration task (i.e., data conversion and transformation). To be specific, we implemented four JOpera adapters (each for a different type of schema mapping) that receive as input the source and target schemas, the schema mapping and matching results, and the value of the output parameters (obtained at runtime). The output of an adapter is, therefore, the data that is

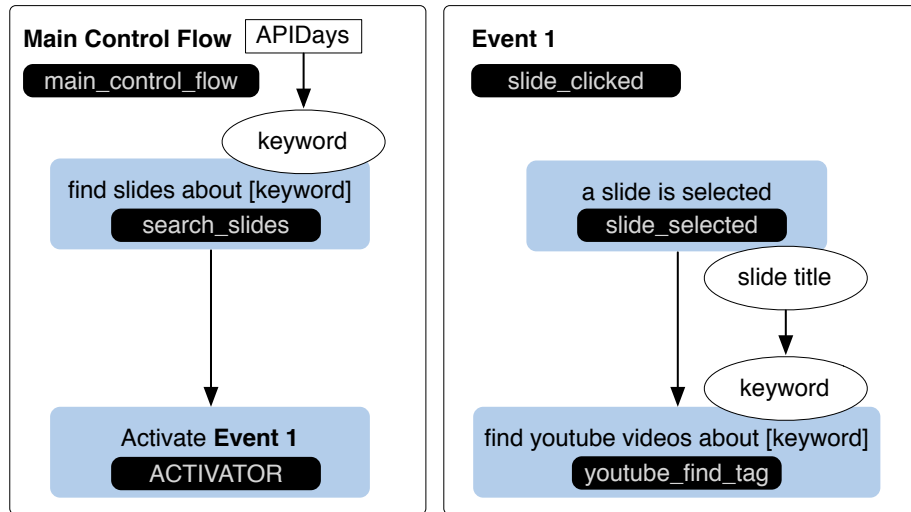


Figure 4.4. The intermediate model generated for Listing 3.1. It contains two control flow graphs: **Main Control Flow** that corresponds to the imperative sentence “find slides about APIDays”, and **Event 1** that is associated with the causal sentence “when a slide is selected, find youtube videos about slide title”. The passing of input data flows to output data flows are represented by ovals.

consumable by the input parameter and matches its schema.

- **Step 5: Intermediate Model** The disambiguated syntax tree is consumed to generate an intermediate model (Figure 4.4) that includes control flow and data flow graphs representing the algorithmic structure of the mashup. This structure includes a “main” control flow graph containing all of the imperative sentences found in the current mashup recipe (and their constituent executable chunks like clauses and phrases built from Event and Task labels) as well as a set of “event-triggered” control flows associated with the causal sentences of the recipe. Causal sentences get activated in the “main” control flow. The “main” control flow begins when the mashup starts executing, whereas “event” control flows are executed every time their corresponding causal sentence occurs. The nodes of these graphs store a mapping between the executable and data elements of the recipe (technology-neutral) and the target executable model of JOpera (technology-specific).

- **Step 6: Emitter** The intermediate model is transformed into the target composition code (Figure 4.5), which is directly executable by JOpera mashup engine, which further transforms it internally to Java bytecode for efficient exe-

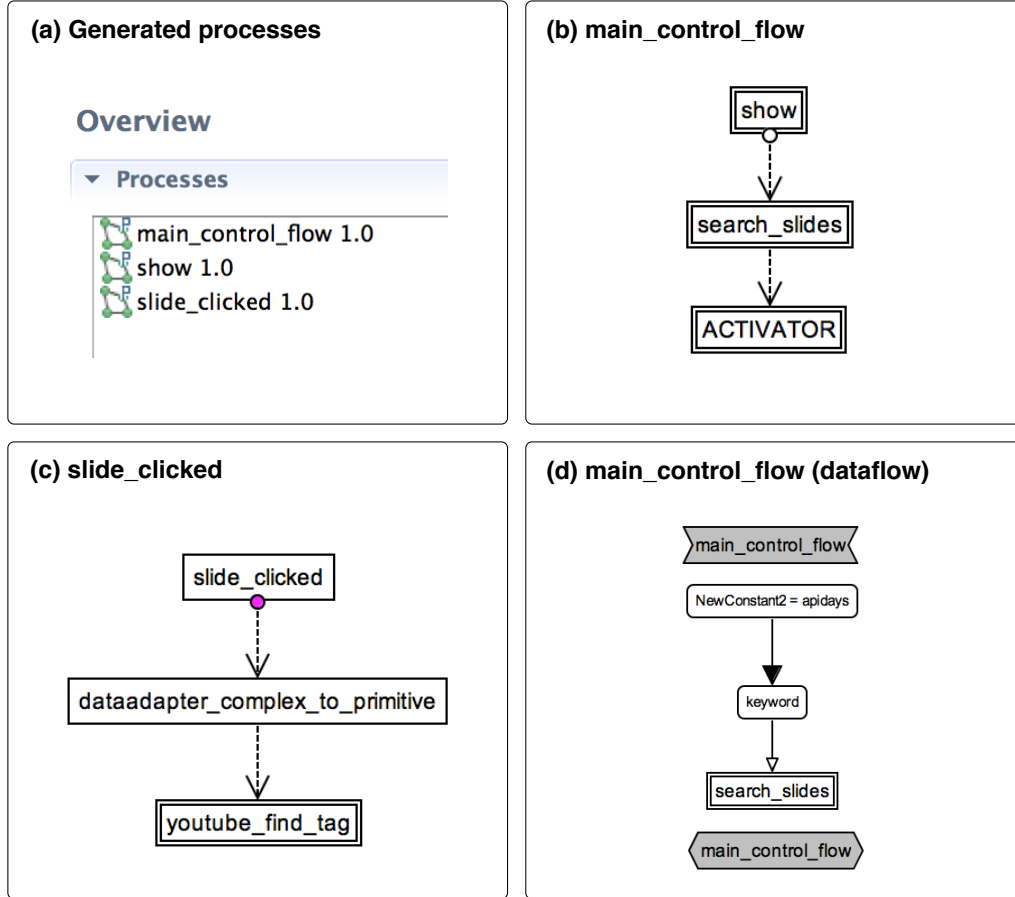


Figure 4.5. The generated JOpera visual composition code [94] for Listing 3.1: (a) list of processes created for the mashup: **main_control_flow** implements the control flow for the imperative sentence, **slide_selected** implements the control flow associated with the causal sentence (slide select event), and **show** is the process responsible for creating the user interface of the mashup, (b) control flow implementing **Main Control Flow** in the intermediate model (Figure 4.4), (c) control flow triggered whenever a slide is selected (it corresponds to **Event 1** in the intermediate model), and (d) data flow associated with **main_control_flow** (“apidays” is a constant passed to the **search_slides** input parameter).

cution. The mashup execution is controlled by *NaturalMash* through a REST API, which also allows to retrieve and display its results. By replacing the emitter it is possible to target other mashup runtime platforms.

The performance of the incremental compilation process is boosted through the use of the *cache* storing all the previous target models and their corresponding generated code. In doing so, the compiler component can save considerable time and computation resources by first looking up the target model in the cache and reusing its corresponding generated code. Also, the cache is populated through crowdsourcing to contain the target models generated not only by the same end-user but also by other concurrent end-users of the system. This way, the chances of the compiler finding the appropriate target model is higher.

4.2.4 Client-server Communication

After the mashup is compiled and deployed, a response is immediately sent back to the client. Since the response is asynchronous, there are three ways for the client to receive it: (i) using HTTP short polling (i.e., polling the server by sending continuous requests in short-intervals), (ii) using HTTP long polling (i.e., keeping the connection open for a long time-interval), and (iii) using Web Sockets. Among these solutions Web Sockets is much faster and more reliable [78], and is therefore, used by *NaturalMash* as the main communication mechanism. Moreover, this technology is nowadays mature and fully supported by the widely-used Web browsers (e.g., Google Chrome, Firefox, Internet Explorer, etc.).

While compiling and deploying the mashup, following response types may be send back to the client via Web Sockets:

- **Error.** The input recipe may contain syntactic and semantic errors. At this point, the client is immediately notified and the compilation process is aborted. Syntactical errors involve trespassing the grammar and syntax of the language, for instance omitting a comma. Semantic errors occur when there is an ambiguity in matching Task/Event labels or describing data flows.
- **Successful compilation.** If the input recipe has no error, it will compile, and a response will be sent to client upon successful compilation.
- **Successful deployment.** After compilation, the deployment process will take place. Once the deployment is completed, a response containing the URL to the resulting mashup will be sent back to the client.

4.2.5 Mashup Runtime

Immediately after the client is notified about the end of the deployment, it proceeds to execute the mashup with the given user input data. With regard to the response time, the most important phase in mashup runtime life-cycle is the first phase, in which the mashup is initiated in the visual field. To optimize this phase we propose two mechanisms based on parallelization and caching.

On the client-side, each widget is initialized in parallel by using WebWorkers to call the associated JavaScript code. The server is responsible for interacting with the ingredients due to the Same-Origin-Policy (SOP) sandboxing limitations imposed by browsers. JOpera provides support for a shared cache of Web service invocation results that can – seen as a form of pre-fetching – reduce the execution time of popular mashups. caching is also a crucial feature in live mashup tools minimizing excessive calls to the ingredients of a mashup, which most probably have some sort of call rate limit that may hinder the continuous re-execution of mashups invoking them. Also, the *NaturalMash* engine is aware of ingredients with call rate limit from the meta-data already provided in the *NaturalMash* component meta-model (it is defined as a property of the interface). Therefore, if a component is about to reach its limit, *NaturalMash* force-freezes its re-execution. Also, we advocate component providers to use OAuth¹, as opposed to developer keys, to identify the source calls. The advantage is that, the call limit is uniquely allocated to each end-user of the mashup tool and is not shared by all the end-users.

¹<http://oauth.net/>

Chapter 5

Evaluating NaturalMash In-The-Lab

NaturalMash evolved over the past two years following a formative user-centered design approach [111], which proposes an iterative and incremental process for design and development of software systems. In the process, each iteration cycle consists of design, implementation, and formative evaluation. The evaluation is conducted at the end of each iteration to inform the next iteration and ensure that users were kept central in the design so to avoid as much as possible mismatches between users' expectations versus system behavior.

We have completed three iteration cycles. In this chapter, we present the results from the formative evaluations we conducted at the end of each iteration, and show how the evaluations have driven the design of *NaturalMash* (Table 5.1).

5.1 First Iteration

The first iteration involved the design, development, and evaluation of the initial prototype of the system (Figure 5.2). For the evaluation, we conducted an expert review with 10 mashup experts. We individually interviewed the experts in order to shed light on existing usability problems, and asked them to define how serious these were. Each expert was asked to interact with the system (after a short tutorial) for as long as they needed to provide feedback. The experts were researchers and practitioners in mashups that we met in mashup-related workshops such as the *5th International Workshop on Lightweight Integration on the Web (ComposableWeb)* and the *6th International Workshop on Web APIs and Service Mashups (Mashups)*. The goal of the review was to identify common usability errors before doing a user study.

Features	V0	V1	V2	Change rationale
Autocompletion	×	×	×	*In addition to the label text, suggestions are represented with the corresponding ingredient icon
Error highlighting	–	×	×	Give immediate feedback about errors to user
Semi-structured editor	–	–	×	Prevent syntax errors
Ingredient stack	×	×	–	Replaced with Ingredient dock.
Ingredient dock	–	–	×	Make the current APIs more visible to users while interacting with the visual and text fields.
Side-bar	–	–	×	Let users tag favorite ingredients (APIs), and retrieve their mashups.
API search box	×	–	–	Users did not realize they could search for APIs.
Inline search	–	×	×	Replaced the search box to let users search for APIs from within the text field.
Ingredients toolbar	–	–	×	Search <i>and</i> browse existing APIs.
Drag & drop	–	–	×	Let users directly use a API selected from the ingredients toolbar.
Programming by Demonstration	–	–	×	Allow users to (visually) demonstrate what they want.
Auto-visualization	–	–	×	Visualizing output data using a suitable widget, thus reducing the complexity of the mashup description with natural language.

Table 5.1. The evolution of NaturalMash during the formative user-centered design process in terms of added/removed features. **V0**, **V1**, and **V2** correspond to the versions of the tool during, respectively, the first, second, and third iterations.

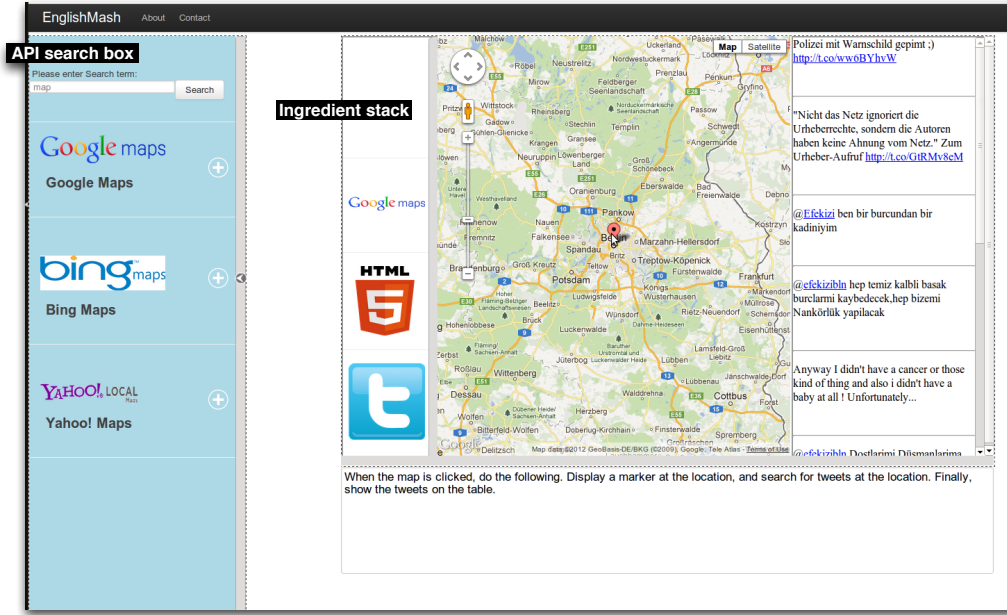


Figure 5.1. NaturalMash environment in the first iteration.

5.2 Second Iteration

The valuable feedback from the expert review informed the re-design of the system. Specifically, the feedback suggested to: (1) remove the search box, that was initially designed as part of the component stack to help with component discovery, and replace it with the inline search functionality (i.e., component discovery within the text field); (2) reorganize the user interface layout (e.g., moving the stack from left to the right); (3) efficiently organize and add icons to the autocomplete menu; (4) provide visual cues to distinguish labels of ingredients already in use out of all returned suggestions within the autocomplete menu; and (5) change short-cut keys for properly working with the autocomplete menu; and (6) improve the environment color scheme and fonts.

At the end of the first iteration, we conducted a user study with the main goal of identifying early major usability problems.

5.2.1 Participants

We repeated the study with two groups of participants with similar profiles, respectively: 5 high school students, and 6 first year students attending the USI Bachelor of Informatics. The recruited high school students had volunteered to

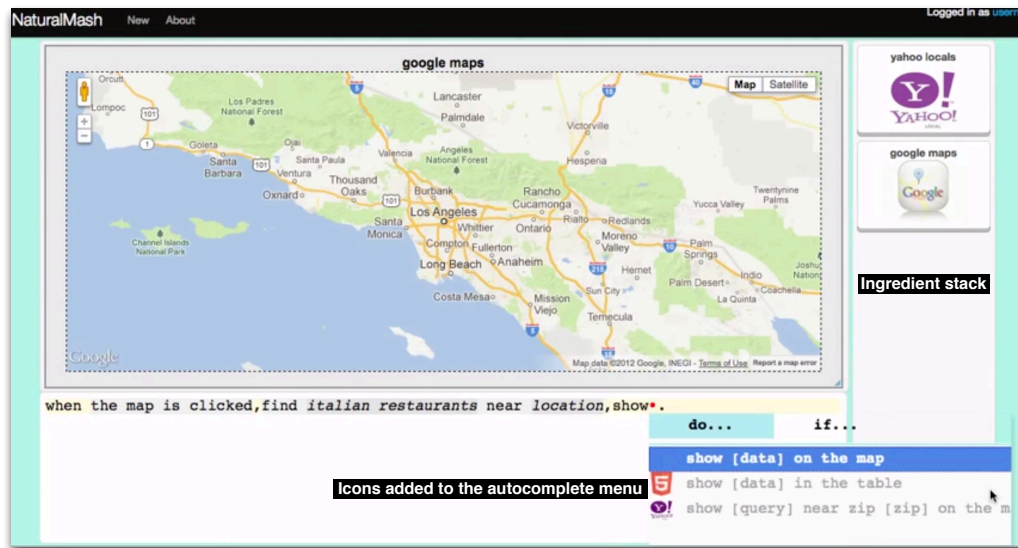


Figure 5.2. NaturalMash environment in the second iteration.

attend a one-week computer science promotion program at our University. We performed the usability testing at the very beginning of the program in order to avoid students to be influenced by any programming activity. The first-year students, who just started their studies at our University, even if inexperienced had an interest in Web technologies and this motivated them to volunteer to participate in the user study. In terms of programming knowledge (see Appendix A, Section A.1), the high school students were all non-programmers. Likewise, among the group of bachelor's students, there were 2 programmers, and 4 non-programmers. Participants in both the groups had neither heard of mashups nor ever created a Web application.

5.2.2 Tasks and Methods

Following Nielsen's discount usability based user testing (<http://www.useit.com/alertbox/discount-usability.html>) we consider the size of our sample sufficient to provide meaningful feedback at this early stage of development. In the beginning of the usability testing, we gave a short tutorial about the system and guided them through completion of a warm-up task that, overall, took around 10 minutes. We then asked the participants to perform five tasks (developing five different mashups) with increasing complexity (in terms of the number of APIs to be mashed up):

Warm-up Task Get upcoming events in a place specified using Google Maps (two APIs).

Task 1 Play YouTube videos selected from Delicious public bookmarks (two APIs).

Task 2 Show Flickr images about the twitter trending topic (two APIs).

Task 3 Aggregate BBC news, CNN news, and Delicious feeds (three APIs).

Task 4 Display tweets and events around a selected map location (three APIs).

Task 5 Create a mashup on your own (open task).

The final open task was designed to assess the ability of the participants to independently come up with a mashup idea, and then transform it to a concrete implementation.

Since the tasks themselves were described in English, we attempted to minimize the similarity of the Task labels with their corresponding solutions in the *NaturalMash* CNL. After the usability testing, we asked the participants to fill out a questionnaire (see Appendix A, Section A.2) to assess their satisfaction with the system as well as to gather their opinions on the overall usability of the system (e.g., their opinion on how helpful or unhelpful the environment features were in the context of the given tasks).

5.2.3 Results

Overall, the participants performed correctly 52 out of 55 tasks. On average, they took less than 5 minutes to correctly complete each task.

82% of the participants believed that the autocomplete and inline search features helped them quickly and easily discover appropriate Task or Event labels. As indicated by 80% of them and according to our observation, the live preview in the visual field was really engaging and had a positive impact on the completion of the tasks. However, correcting mistakes in the system was difficult for 55% of them.

5.2.4 Lessons Learned

The evaluation helped to identify early critical usability problems. The freestyle editor in the text field makes it easy for the participants to make syntax errors, which in turn causes anxiety and stress. Another usability problem, revealed in the open task, concerns the lack of an overview of the available ingredients.

More importantly, most participants expected that the results of their commands would have been displayed immediately. Instead, in order to display the results of a Task on the visual field (e.g., “find tweets around location”), they had to learn how to manually find the right widget and explicitly add its corresponding Task to the mashup recipe (e.g., “show the result in the table”).

5.3 Third Iteration

In this iteration, we attempted to address the usability problems identified in the previous iteration by implementing a set of feature additions to *NaturalMash*:

- The semi-structured editor, making it unlikely to make syntax errors, while allowing freestyle editing.
- The ingredients stack that gives a searchable overview of existing ingredients and the corresponding drag and drop support. We extended the component library by adding 15 more popular ingredients to enable innovation in the open task (it used to contain 7 ingredients in the first iteration).
- The automatic visualization of the results in a widget (e.g., table, map, or chart) without end-users having to explicitly mention the widget.

We also moved the component dock from the right side (where the stack currently is) to the top of the text field to be more visible while end-users interact with the text field or visual field. After observing some participants dragging the icons of some ingredients over the widgets in the visual field we decided to add support for PbD in the second version.

To conclude this iteration, we conducted a formative evaluation on a larger group of participants. The primary goal of the evaluation was to identify usability problems, but also to assess the success of the new features in addressing the usability problems identified in the previous iteration.

5.3.1 Participants

We recruited a total of 22 participants, mostly from young university staff and students volunteers both at the University of Lugano and at the University of Trento. In terms of programming skills (see Appendix A, Section A.1), they were equally divided into programmers and non-programmers.

5.3.2 Tasks and Methods

The participants were given four tasks of growing complexity (in terms of the number of APIs to be mashed up), after receiving a short tutorial (5 minutes) in the form of a warm-up task (with the complexity of two APIs):

Warm-up Task Get upcoming events in a place specified using Google Maps (two APIs).

Task 1 Search Flickr images with location from Google Maps (two APIs).

Task 2 Show upcoming events in a selected location on the map. Get information about each event from Google (three APIs).

Task 3 Find slides about “Web APIs”. For each slide found, show relevant videos, tweets, and images (four APIs).

Task 4 Create a mashup on your own (open task).

During the study we recorded the session (video, audio, and screen) and asked the participants to think aloud about their activities. The recordings were complemented by an informal interview as well as an exit questionnaire (see Appendix A, Section A.3) asking them about their overall reaction and satisfaction with the system. The aim of the extended post-study interviews was to have a deeper, but informal discussion with each participant with the opportunity to reflect on what was not captured by the questionnaires and to further discuss the rationale behind some answers. For instance, we asked “do you have any personal mashup/use-case?” and “why do you feel comfortable with the system?”.

5.3.3 Results

In terms of accuracy and efficiency (Figure 5.3 and 6.4), the majority of participants completed all the tasks correctly and in a very short time (around 3 minutes on average), with a slightly better performance on the programmers’ side. Out of the total of 88 tasks, 11 programmers produced 42 correct tasks, while the 11 non-programmers achieved 40 correct tasks. Moreover, by the end of each user study session, the majority of participants felt confident about their mastery of the system and reported on a high level of satisfaction in the exit survey (72% felt satisfied with the system, 89% were interested in continuing using it, and 87% wanted to suggest it to their friends).

Our recorded observations together with the feedback from the participants through both the exit questionnaire and informal discussion, reported positively

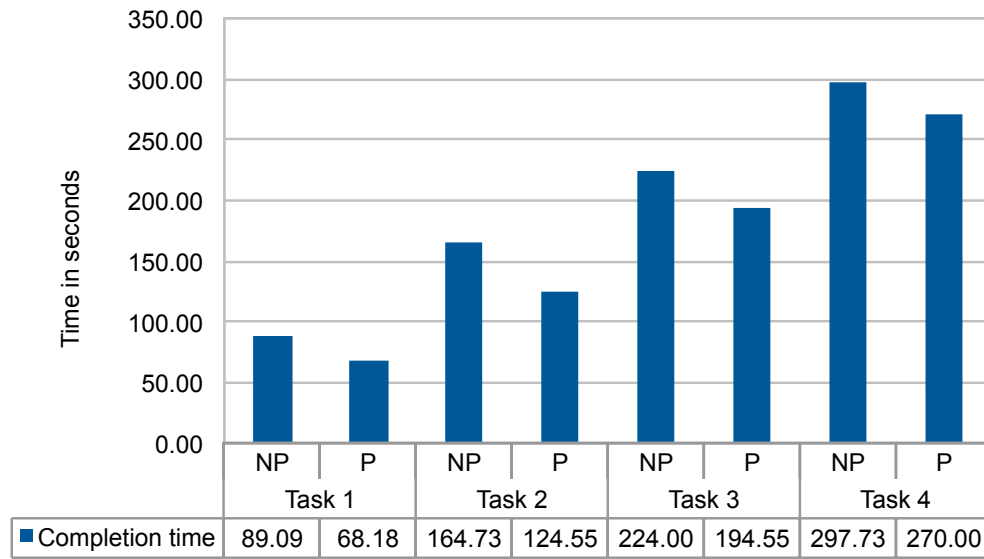


Figure 5.3. The completion time grows with the complexity of the task at hand. Programmers have a slightly shorter completion time than non-programmers.

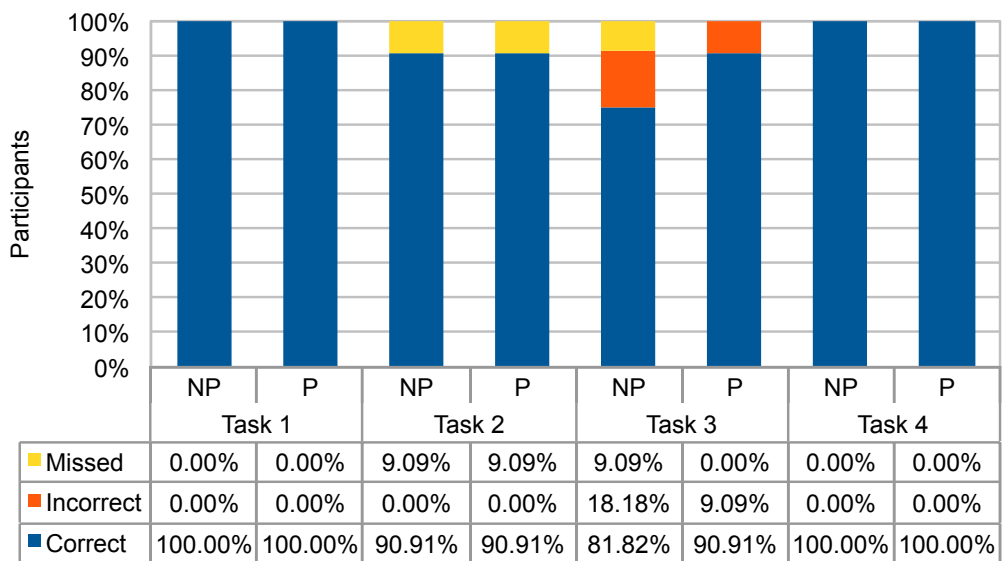


Figure 5.4. The majority of tasks were accomplished successfully (correctly). In terms of accuracy, however, there is no major difference between programmers (P) and non-programmers (NP).

on the individual features of the *NaturalMash* environment (see Chapter 3, Section 3.5). More in detail, the following percentage of participants reported that the features were helpful or very helpful for the completion of the given tasks: autocomplete (92%), inline search (91%), live execution (86%), ingredients toolbar (82%), and PbD (73%). The ingredients toolbar was more frequently used for component discovery and selection than the inline search with the text field (in average, 84% of component discovery and selection tasks were done using the ingredients toolbar). Instead, the participants employed the inline search feature when they were looking for a specific operation that could be perfectly described verbally.

Overall, the participants were engaged with the system (77% felt the system was stimulating or very stimulating). In the open task, all created distinct, useful, and non-trivial mashups. One example was a mashup that finds an audio album in eBay and plays it in YouTube by first searching and finding the exact name of the album using the Last.fm API. Another example was a mashup that shows news, Flickr images, and tweets all related to a selected location on the map, and then allows to share the results on Facebook. Indeed, some of the mashups created in the open task were actually meant to address a real pressing need of the participants. For instance, one of the participants created a mashup to automate the analysis of online presence within the tourism domain. The mashup searches tweets for a specific tourism-related keyword, and then for each tweet found, it searches for the Facebook profile given the name of the author of the tweets.

Lessons Learned

We observed that some participants – especially non-programmers who lack algorithmic thinking abilities – would benefit from receiving suggestions not only for individual Event/Task labels but also for hints on how to compose them together in the right order.

Another major usability problem is concerned the way PbD is applied in the visual field, i.e., interacting with widgets results in the corresponding Event label being added to the text field. However, many participants confused capturing the general behavior of a widget (i.e., the event) with the recording of the concrete action on the widget they just triggered (e.g., a specific location they have clicked). For example, clicking on the map would add the map-click Event label (“when the map is clicked”) to the text. This is meant to be completed by appending Task labels (e.g., “show upcoming events around location”) that form the body of the causal sentence. Indeed, we observed – in the same map

example – the participants correctly generating the Event labels using PbD and completing it with an imperative clause, expected to immediately see the results from the location they had originally selected (demonstrated) to create the causal sentence, as opposed to having to click again on the map to obtain the results. In other words, they did not realize they had created a parametric mashup that shows events for any location on the Map. A similar problem occurred with other widgets supporting PbD, such as the table.

Chapter 6

Summative Evaluation of NaturalMash

As discussed already, one of the goals of *NaturalMash* as an optimal usable mashup tool is to ensure a high level of usability by non-programmers (we will discuss expressive power in the next chapter). Throughout the user-centered design of the system, we conducted three formative evaluations to identify the usability problems hindering this goal (Chapter 5). In this chapter, we comprehensively assess the attainment of this goal by measuring how usable *NaturalMash* is to its potential end-users.

6.1 Evaluation Questions and Hypotheses

The evaluation seeks to answer the following questions:

- **RQ1** How usable the system is for end-users with different programming backgrounds?
- **RQ2** How do end-users with different programming backgrounds interact with and feel about the features of the system (i.e., the text field, the visual field, and the ingredient toolbar)?
- **RQ3** How much can the system help its end-users to create the mashups they need?

We also consider the following hypotheses in relation to the above questions (each hypothesis corresponds to the similarly numbered research question):

- **H1** The the level of the usability experienced by end-users is independent from their programming background.

- **H2** For API discovery tasks, programmers tend to use the text field more frequently than the ingredient toolbar. The situation may be opposite for non-programmers.
- **H3** The major user-perceived limitation of the system is the lack of required ingredients.

6.2 Evaluation Methods Overview

In order to meet the evaluation goals and comprehensively address the stated research questions, the evaluation should be summative. A summative evaluation, as opposed to previously conducted formative evaluations, judges the system as it is and makes statistically significant statements about it. Therefore, a summative evaluation involves a relatively larger number of subjects than, for instance, a formative evaluation. Lab usability testing methods used in previous formative evaluations, despite their strengths in identifying major usability problems, are time-consuming and thus cannot be easily scaled to large number of participants. For these reasons, the current summative evaluation uses *online usability testing* methods, in which the test is conducted remotely (the participant and moderator are not in the same physical location), and asynchronously (it is not “live” with a moderator). These characteristics of online usability testings allow to test many participants simultaneously, and therefore to collect larger samples of both qualitative and quantitative data in a short time.

6.3 Data Collection Methods

In this evaluation, we collected both qualitative and quantitative metrics (Table 6.1). These metrics are collected either automatically or in a self-reported manner, and are categorized into task-based and end-of-session groups.

6.3.1 Task-based Metrics

Task-based metrics give detailed information about task effectiveness and efficiency.

- M1** *Task status*. This metric provides three status information for tasks: (a) *Correct*, (b) *Incorrect*, (c) and *Skipped*. The participants self-report “skipped” status for a task or indicate that it is complete by clicking, respectively,

Metrics	Description	Type	Research questions
Task status (M1)	The status of a task as <i>Correct</i> , <i>Incorrect</i> , or <i>Skipped</i> .	Self-reported	RQ1
Task time (M2)	For each participant, the time spent on the tasks	Automatic	RQ1
Efficiency (M3)	Number of correct tasks per minute	Automatic	RQ1
Clickstream (M4)	Captures mouse activities (e.g, movements and clicks)	Automatic	RQ2
Activity logs (M5)	Number of using different discovery methods (M5.1) as well as the rate of using permitted keys in the text field (M5.2)	Automatic	RQ2
After task rating scales (M6)	Measuring perceived ease-of-use (M6.1) and usefulness (M6.2)	Self-reported	RQ2, RQ3
After task open-ended questions (M7)	Qualitative explanation after the skipped tasks	Self-reported	RQ2, RQ3
Comments (M8)	Arbitrary feedback while doing the tasks	Self-reported	RQ2, RQ3
Satisfaction (M9)	The system usability scale (SUS) to assess the user satisfaction with the tool	Self-reported	RQ2, RQ3
End-of-session open-ended questions (M10)	Overall qualitative feedback about the tool	Self-reported	RQ2, RQ3

Table 6.1. The metrics used in this evaluation to assess the evaluation questions and hypotheses (Section 6.1 on page 77). They are collected either automatically or in a self-reported manner.

the “Skip” and “Finish” buttons (Figure 6.2). Whether the tasks marked as completed are “correct” or “incorrect” is later assessed by the moderator.

- M2** *Task time.* For each participant, the metric is the amount of time spent on both the skipped and completed tasks.
- M3** *Efficiency (correct tasks per minute).* This metric combines both task status and time metrics into a single measure. For each participant, it is calculated by dividing the total number of correct tasks by the time spent on all the tasks (in minutes).
- M4** *Clickstream data.* It captures and logs mouse activities (e.g, movements and clicks) on different parts of the user interface.
- M5** *Activity logs.* This set of metrics provide a fuller picture of how individual participants behave and interact with different parts and features of the system user interface. They are measured at both task time and session time interval.
 - M5.1** *Number of using different discovery methods.* As it was mentioned in the design chapter, *NaturalMash* provides two different methods for API discovery. Name-based discovery using the toolbar and functionality-based discovery using the text field inline search feature. This metric captures how many times each of the two methods has been used.
 - M5.2** *Number of permitted and non-permitted keys used in the text field.* The text field provides a semi-structured editor, meaning that not all keys are permitted to be used. This metric is the frequency of using permitted keys vs non-permitted keys.
- M6** *Rating scale metrics:* Self-reported metrics are captured after each task is done.
 - M6.1** *Ease of use.* It shows the degree of perceived ease-of-use for different features of the system.
 - M6.2** *Usefulness.* It suggests how useful different features of the system are to the participants.
- M7** *open-ended questions.* After a task is skipped, open-ended questions seek to collect qualitative explanation.
- M8** *Comments.* Participants are also allowed to make arbitrary comments while performing the tasks.

6.3.2 End-of-session Metrics

These metrics are collected at the end of the session (after conducting the tasks) through self-report questionnaires.

M9 *Satisfaction.* The system usability scale (SUS) is a ten-item Likert scale showing the overall attitude of the participants towards the system.

M10 *open-ended questions.* Overall qualitative feedback about the the system is collected using open-ended questions.

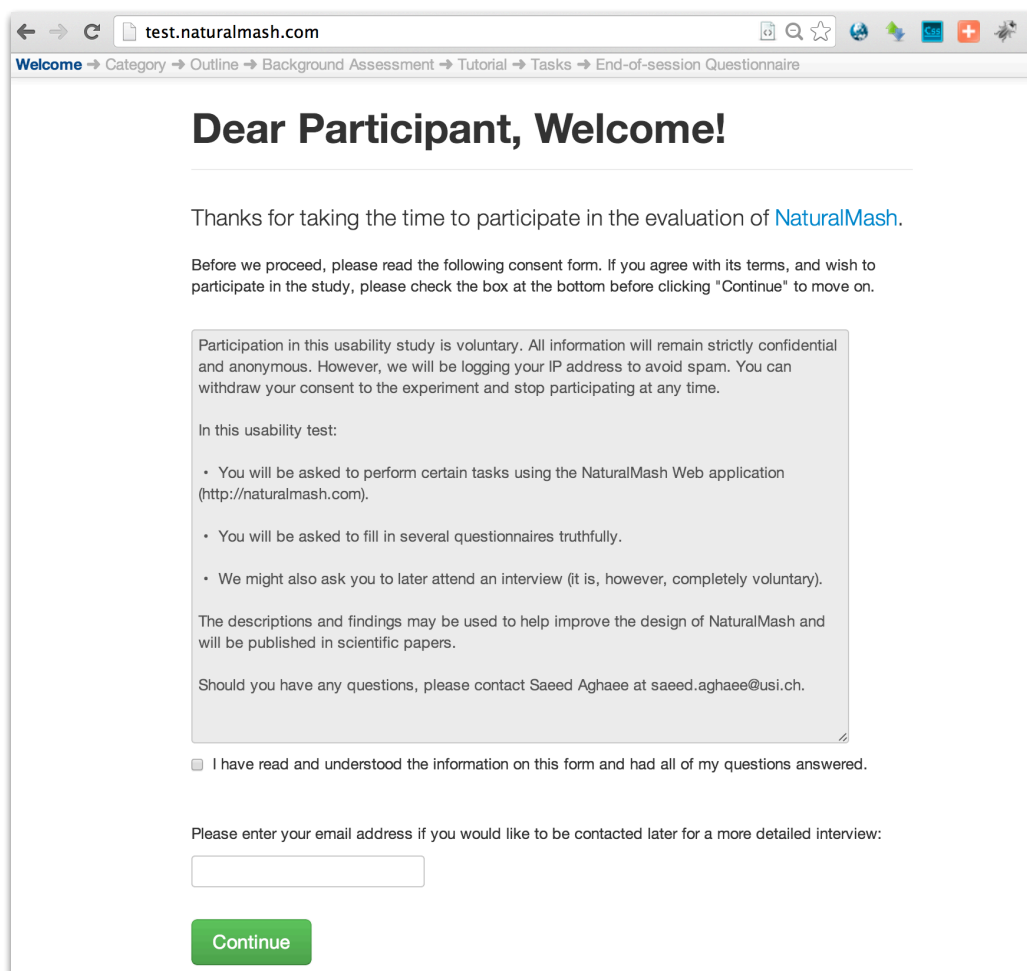
6.4 Technology

We developed a website (available at <http://test.naturalmash.com>) that self-contains all the necessary information and functions to carry out an unmoderated remote usability evaluation. Figure 6.1 and 6.2 illustrate two snapshots of the website. The website is powered by a central database (MongoDB) storing all the metrics data mentioned above, except for activity logs. The system itself is responsible for collecting the activity logs metrics and storing them in the database. All the data is tied together with unique participant IDs generated by the website. In addition to the evaluation process, we also fully automated the data analysis task. We created a script that fetches the data from the database, analyzes it, and subsequently generates Excel files containing the results.

6.4.1 Participant Recruitment and Sample Collection

In this study, we recruited a sample of 40 participants with different programming backgrounds. The sampling mechanism made sure that we have enough and equal number of participant for each group of users (i.e., 20 programmers and 20 non-programmers). Since we did not have access to all the potential users of the system (adult Web users), we used a convenience non-probability sampling method, in which subjects are selected because of their background (programmer or non-programmer) and the fact that they are available to participate in the study.

We used two different recruitment methods to assemble the participants (Figure 6.3). Part of the participants were recruited by direct invitation. They had already registered in the main website of the system, or had been personally asked to join the user study (they were mostly from first year bachelor students



The screenshot shows a web browser window with the address bar displaying "test.naturalmash.com". The browser's navigation bar includes a back arrow, a forward arrow, a refresh button, and icons for search, star, and social media. Below the address bar is a breadcrumb trail: "Welcome → Category → Outline → Background Assessment → Tutorial → Tasks → End-of-session Questionnaire".

The main content area has a large heading "Dear Participant, Welcome!" followed by a horizontal line. Below this, the text reads: "Thanks for taking the time to participate in the evaluation of [NaturalMash](#)."

Next is a paragraph: "Before we proceed, please read the following consent form. If you agree with its terms, and wish to participate in the study, please check the box at the bottom before clicking 'Continue' to move on."

A light gray box contains the following text:

Participation in this usability study is voluntary. All information will remain strictly confidential and anonymous. However, we will be logging your IP address to avoid spam. You can withdraw your consent to the experiment and stop participating at any time.

In this usability test:

- You will be asked to perform certain tasks using the NaturalMash Web application (<http://naturalmash.com>).
- You will be asked to fill in several questionnaires truthfully.
- We might also ask you to later attend an interview (it is, however, completely voluntary).

The descriptions and findings may be used to help improve the design of NaturalMash and will be published in scientific papers.

Should you have any questions, please contact Saeed Aghaee at saeed.ghaee@usi.ch.

Below the gray box is a checkbox with the text: "I have read and understood the information on this form and had all of my questions answered."

Below the checkbox is a text input field with the label: "Please enter your email address if you would like to be contacted later for a more detailed interview:"

At the bottom is a green button labeled "Continue".

Figure 6.1. The welcome screen of the evaluation website. It contains a welcome message and a consent form.

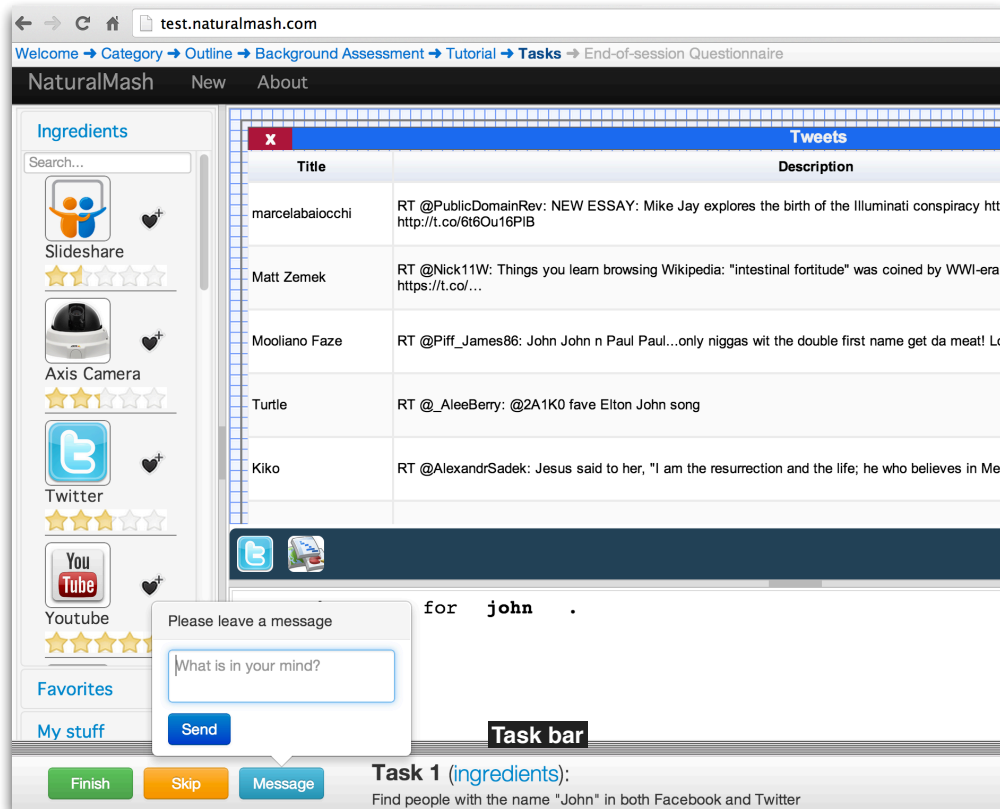


Figure 6.2. This is how NaturalMash is presented to participants in the evaluation website. It contains an IFrame that shows NaturalMash and one auto-hide Task Bar that includes the task description and allows participants to “Finish” or “Skip” the tasks as well as leave a “Message” reflecting what they think while doing the task.

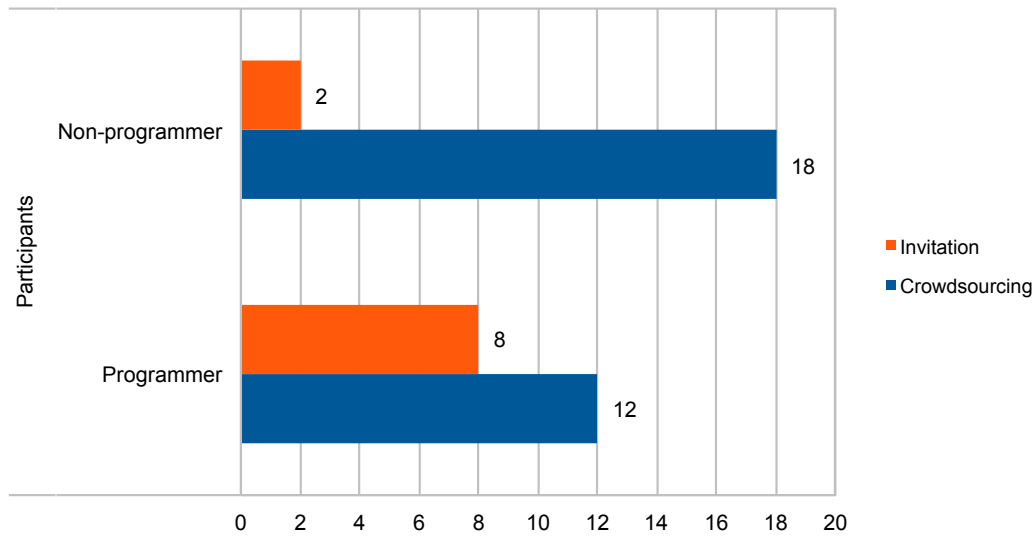


Figure 6.3. The number of participants recruited by each method (invitation, and crowdsourcing).

in the University of Lugano). The rest (30) were directly recruited from crowdsourcing websites (e.g., microWorkers) and incentives were provided in the form of money (2 U.S. dollars).

6.5 Evaluation Procedure

6.5.1 Starter Questions

The starter questions (see Appendix A, Section A.1) assess the participants background in terms of (i) knowing how to program, and (ii) the the level of awareness about different services on the Web (e.g., LinkedIn, Google Maps, etc.).

6.5.2 Tasks

Before conducting the tasks the participants are given a short tutorial. The tutorial is a 9 minutes video in which the system is explained to participants through demonstrating a warm-up task. The warm-up tasks is about creating a mashup that helps with finding the music videos of songs wit the title “remix”, and also allows to share the music videos on Facebook. The mashup uses Last.fm API, Facebook API, and Youtube API.

The evaluation consists of 4 tasks, in each of which the participants are asked to create a mashup. 3 out of the 4 tasks are pre-constructed and 1 is self-generated (open task). The participants are asked to self-generate the open task (writing the text description in natural language) after completing the pre-constructed tasks. The goal of the open task is to assess how useful the system is in real situational use-cases suggested by the participants.

The pre-constructed tasks are sub-selected from a pool of 9 tasks, equally divided into three classes:

Class 1 This class of tasks aims at inspecting the usability of the name-based discovery method by deliberately including the list of to-be-used components in the task description. As a result, we expect the participants to more frequently use the toolbar to search for and then drag and drop the required components to complete the task. We pre-constructed three tasks in this class with the same level of complexity as follows:

Task 1.1 “Show recent images and videos about PS4.”

(list of to-be-used component: Flickr, and Youtube)

Task 1.2 “Show tweets and Google search results about singularity.”

(list of to-be-used component: Google, and Facebook)

Task 1.3 “Find people with the name John in both Facebook and Twitter.”

(list of to-be-used component: Twitter, and Facebook)

Class 2 The functionality-based component discovery is examined by this class of tasks, whose descriptions does not come with a list of to-be-used components. Thereby, the participants are led to use the inline search feature to discover the needed components based on their functionality. Following tasks are designed for this class:

Task 2.1 “Get news and upcoming events from Switzerland.”

(List of components not given)

Task 2.3 “Get recent tweets and news from Switzerland.”

(List of components not given)

Task 2.3 “Find macbook pro products and images.”

(List of components not given)

Class 3 Another important feature of the system is the possibility of using PbD in the visual field to generate event labels. This feature is mainly tested in

by tasks of this class, where the to-be-used components are mostly widgets that should be chained to each other through their event behavior. The tasks are as follows:

Task 3.1 “Show slides about Web APIs, and then get related videos for each slide found.”

(list of to-be-used component: Slideshare, and Youtube)

Task 3.2 “Get software engineer jobs in Switzerland, and then allow to share the results in Facebook.”

(list of to-be-used component: Linkedin, and Facebook)

Task 3.3 “Get music albums with the keyword “mashup” in the title, and then show related images for each album found.”

(list of to-be-used component: Last.fm, and Flickr)

The sub-selection process is based on the participants’ answers to the Web API familiarity questions in the beginning of the study (starter questions). The three sub-selected tasks each fall into a different class.

Like the previous formative evaluations, in order not to lead the participants, we tried to avoid any similarity between the task descriptions and their corresponding solution in the *NaturalMash* CNL. However, we did not design and sort the tasks based on their level of complexity. Instead, the tasks in this study each probe specific features of the system user interface. The reason for these decisions lies in the shift in our evaluation objective: from formative to summative, or in other words, from receiving general feedback on the major usability issues that have a high chance to be discovered through increasing the complexity of the tasks, to comprehensively measuring the usability of different components of the system user interface.

6.5.3 Post-Task Questions

The post-tasks questions (see Appendix A, Section A.5, Section A.6, Section A.7, and Section A.8) are asked straight after the completion of each task and reflect the experience of the participants with the system. These questions are different for each class of tasks depending on what part of the user interface the task is intended to assess. For instance, the post-task questions for the class 2 tasks tap into the usefulness and usability of the autocomplete menu. Additionally, in case of skipping a task, the participant is required to explain why (open-ended question).

6.5.4 Post-Session Questions

The exit questionnaire (see Appendix A, Section A.4) consists of the SUS as well as an open-ended question asking the participants for general feedback about the system.

6.5.5 Wrap-up

The last page is set as a “Thank You” page, where we also ask participants to give use emails of those who might be interested in doing the evaluation. The page also contains a confirmation ID (built from the unique ID of the participant) which is useful if the participant has been recruited via crowdsourcing. In this case, the confirmation ID is used as a proof that the work (completing the user study) is done by the participant.

6.6 Results

In the following we present and analyze the data collected from the study. We mainly performed segmentation analyses by slicing up the data based on the groups of participants (i.e., programmer and non-programmer) and tasks (i.e., three classes of tasks plus the open tasks). This type of analysis helped us gain a deep insight into the usability of the system.

6.6.1 Task Performance Data

In this section, we only consider the data collected from the tasks of the three classes. We postpone the analysis of the open tasks to the next section. Out of 120 tasks, 7 were skipped, and the rest (113) were reported finished. From the finished tasks, we then assessed 100 as “correct” and the rest (13) as “incorrect”. A task is graded as “correct” if the produced mashup does what the task description says, otherwise the task is scored as “incorrect”. We obtained an average accuracy (percentage of correct tasks) of 83% (i.e., 83% of the tasks were scored as “correct”). Looking into the segments of the data obtained by the three classes of tasks (Figure 6.4), the average accuracy value, however, differs from one class to another. The highest and lowest average accuracy values are for the tasks of, respectively, class 1 and class 3, which are also significantly different (chi-square test with confidence 95% and $p = 0.006$).

The average completion time for all the correct tasks and for all the participants is less than 3 minutes (Figure 6.5). The class 3 and class 1 tasks have,

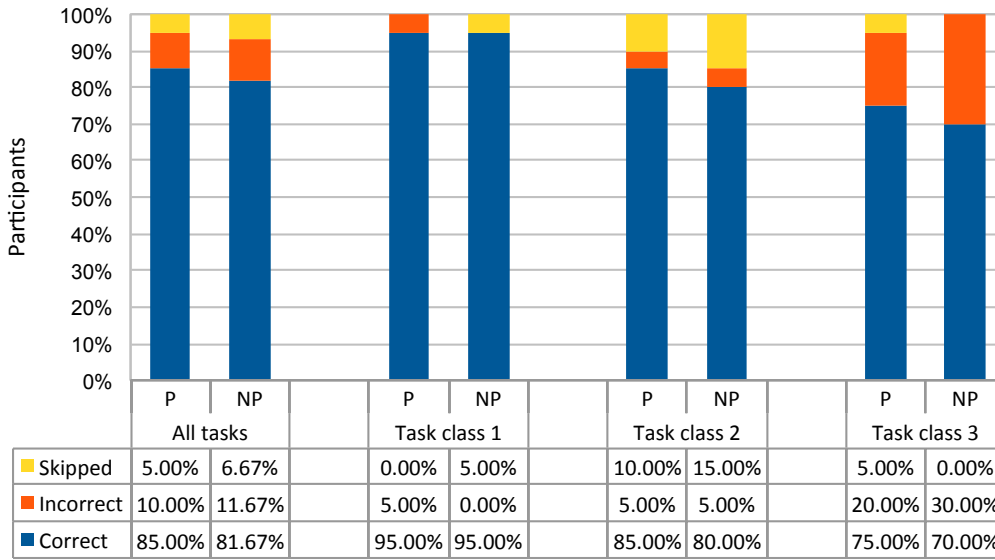


Figure 6.4. Average accuracy of the pre-constructed tasks. **P** = Programmers, **NP** = Non-programmers.

respectively, the longest and shortest completion time (both correct and overall). In general, programmers have spent moderately less time on completing the tasks (both correct and overall).

Figure 6.6 shows the efficiency metric combining task success rate with completion time into a single measure. This metric allows us to more easily compare the performance of programmers with non-programmers. To this end, we conducted a TOST (Two One-Sided Test) equivalence test on the two sample groups (confidence of 95% and equivalence range of 0.15) that produced $p = 0.044 < 0.05$ and $p = 0.017 < 0.05$. The p numbers imply that the two groups have equal efficiency in the range of ± 0.15 .

Open Task

The participants self-generated many interesting tasks. Below we list some of them:

Open task example 1: *“I want to plan a travel between a certain list of places and then compute the shortest path between these places and display it on a map with some images of each place.”*

Open task example 2: *“I want to collect the top songs and then create a mashup song out of them.”*

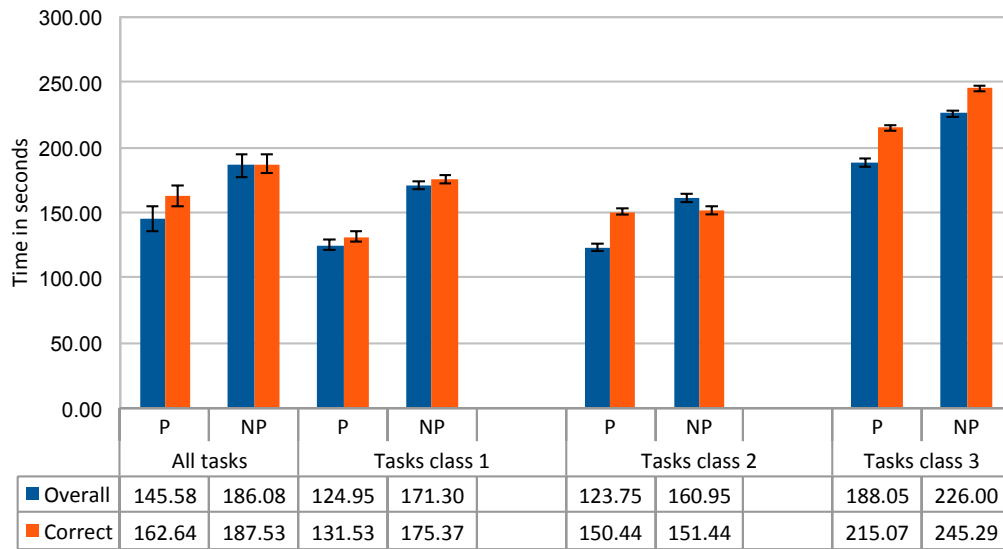


Figure 6.5. Average completion time (in second) for the pre-constructed tasks. **P** = Programmers, **NP** = Non-programmers.

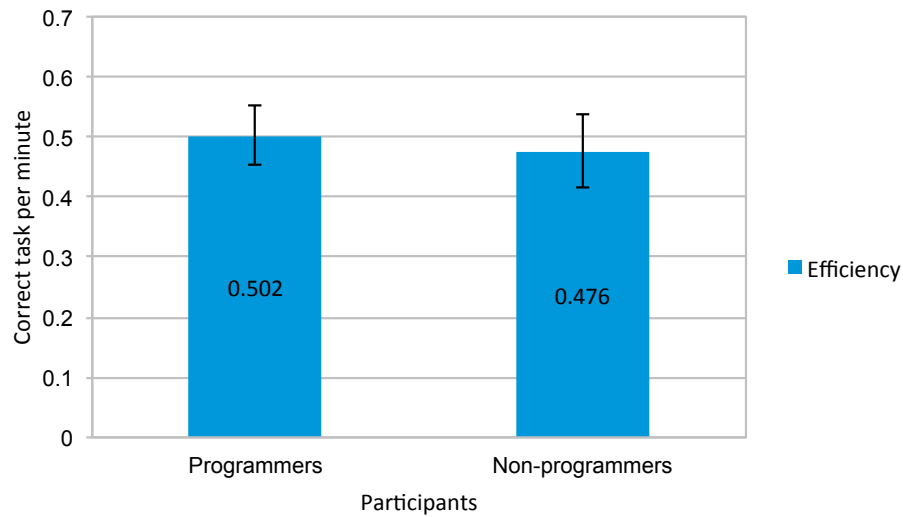


Figure 6.6. Number of “correct” tasks per minutes (efficiency) for programmers, non-programmers, and both (overall).

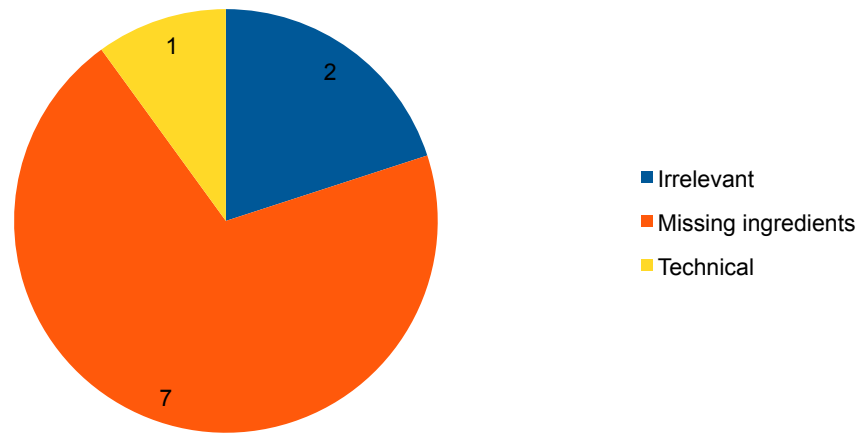


Figure 6.7. The participants' reasons for skipping the open task.

Open task example 3: *“I would like to look for a love song and find it on Youtube then share in my twitter account.”*

We assessed the open tasks in the same way as the pre-constructed tasks. We attained that the average accuracy of the open task for all the participants is 60%. Looking in more detail, the average accuracy for programmers and non-programmers were, respectively, 50% and 70%. Interestingly and in contrast to the pre-constructed tasks, programmers scored notably a lower accuracy and produced more “incorrect” tasks. After manual analysis of the participants' explanation for skipping the open tasks, we concluded that their main reason was the lack of required ingredients in the library (Figure 6.7). Another reason claimed by only two participants was concerned with bugs in the system (e.g. crashing, freezing, etc.). For instance, below is a sample answer provided by one of the participants to the post-task question asking why the participant skipped the open task:

“There was not an ingredient for movies so I cannot complete the task”

Also in case of the “incorrect” tasks, inspecting the provided solutions as well as the comments left while doing these tasks, led us to arrive at the same reason (lack of required ingredients). Here is a sample comment left by one of the participants:

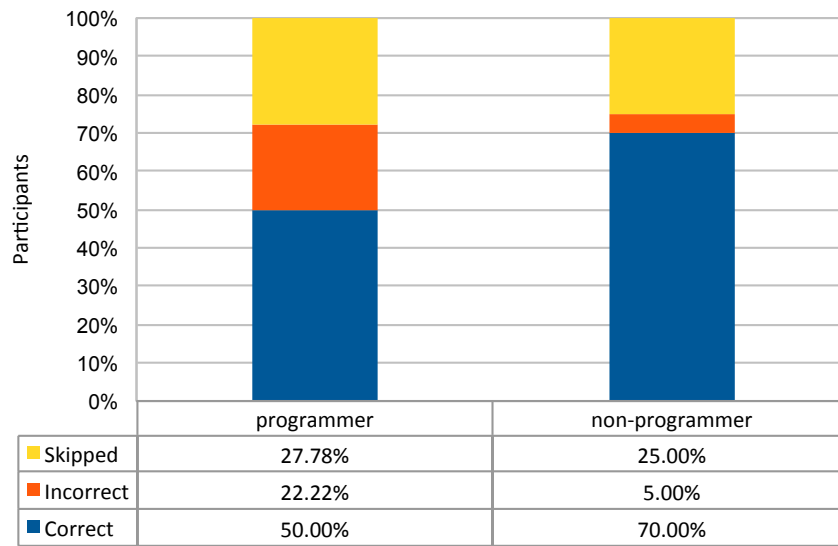


Figure 6.8. Accuracy of the open tasks for programmers and non programmers as well as overall participants.

“I couldn’t get the video location, and i don’t think i can entirely do what i wanted, so i just switched to search video, display it in lugano and search tweets for the video title.”

The average completion time was overall around 60 seconds (1 minutes), and 80 seconds for “correct” tasks. These average completion times are particularly lower than those of the pre-constructed tasks.

6.6.2 Self-reported Data

The post-task questions were designed to objectively assess the ease-of-use and usefulness of the system. Figure 6.9 illustrate the perceived ease-of-use of the three main features of the system: the autocomplete menu (for selecting and adding labels), the ingredient toolbar (for selecting and adding ingredients), and the text field (for editing). The three features mentioned above received different scores, whose mean is 2.9 (maximum is 4). After analyzing the data we learned that autocomplete menu and ingredient toolbar were perceived to have a (marginally) significantly different level of ease-of-use (independent one-tailed t-test with confidence 95% and $p = 0.05$).

Figure 6.10 shows the perceived usefulness of the autocomplete menu, the live and interactive visual field, PbD to generate Event labels, and the text high-

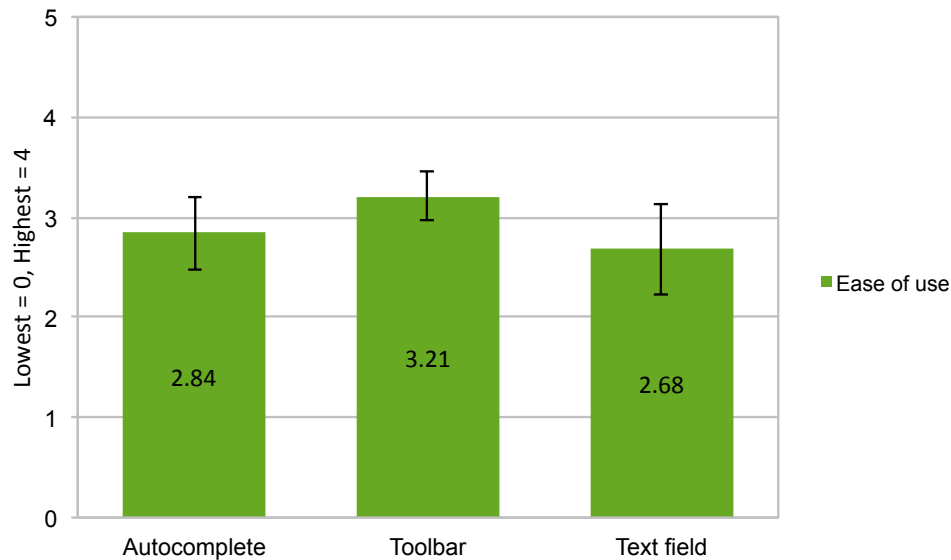


Figure 6.9. Perceived ease-of-use for different features of the system.

lighting feature for data flows and errors. These features were perceived to be useful with the average total score of 2.8 out of 4. The live and interactive visual field was ranked the lowest.

The average score of the SUS questionnaire was 82, in a scale of 0 to 100. Programmers and non-programmers yielded slightly different SUS scores (80 and 83, respectively).

The optional open-ended question was answered by 25 participants. The answers included praise, constructive criticism, and new ideas. Below are some of these answers from the crowdsourced participants:

- “Nice project hope you’ll launch it soon, I must say that at the first time it looked so boring but after the first task all things started to connect one with the other.”
- “I do not like the fact that there are too many pop up windows.”
- “I think it will be a great tool for any normal user like me if it had a more graphical environment. I would use it daily if it acts as a browser too. I need to hide those icons on the left sidebar and use the software as a browser. when required, I can just unhide them and make use of the functionality. I think it would be a wonderful tool for me if it was integrated with free google chrome browser. Anyway the idea behind this work is really appreciable. I believe it would be a great tool in the future for a lot of people.”

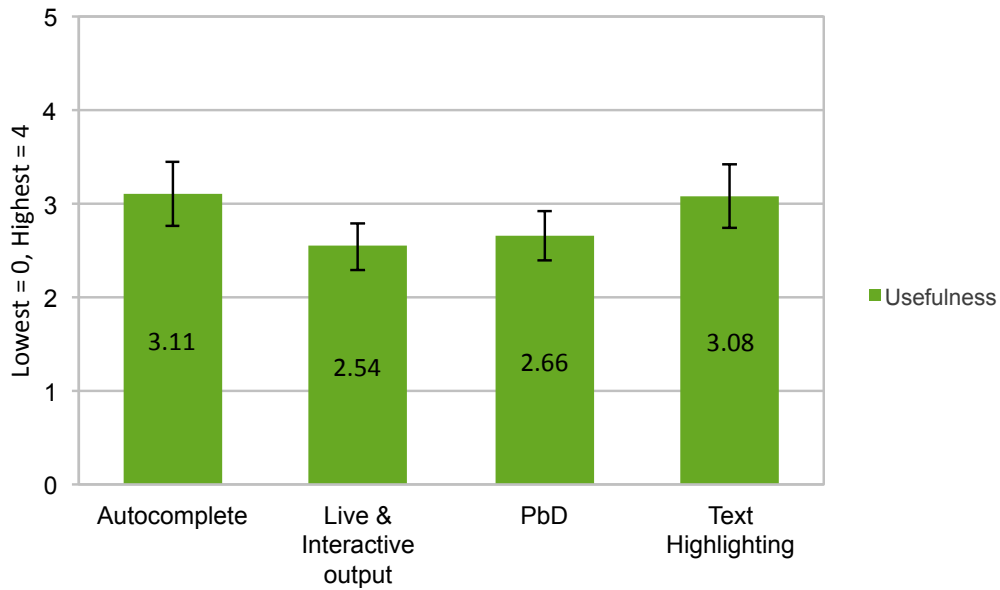


Figure 6.10. Perceived usefulness for different features of the system.

6.6.3 Logs

We collected a large amount of activity logs in accordance with the metrics we mentioned previously. One of the metrics was the percentage of not-permitted keystrokes relevant to all the keystrokes captured in the text field. In average, the value of this metric is around 8.87% for all the participants, 9.40% for programmers, and 8.47% for non-programmer (Figure 6.11).

Looking deeper into the used-keys data, Delete keys (i.e., Backspace and Del) were the most used not-permitted keys. With a big difference, Enter was the second most used not-permitted key. The rest of the used not-permitted keys without significant frequency were comma (“,”), Dot (“.”), and Backslash (“\”). Also, as depicted in Figure 6.12, the top 2 used not-permitted keys for both programmers and non-programmers are the same.

Another collected metric was the ratio of using the two different Web API discovery methods (i.e., functionality-base and name-based). The results are depicted in Figure 6.13. The functionality-based discovery method was utilized more frequently in the tasks of class 2. Moreover, as it is shown in Figure 6.14, programmers used functionality-based method more often than non-programmers (chi-square test with confidence 95% and $p = 0.008$).

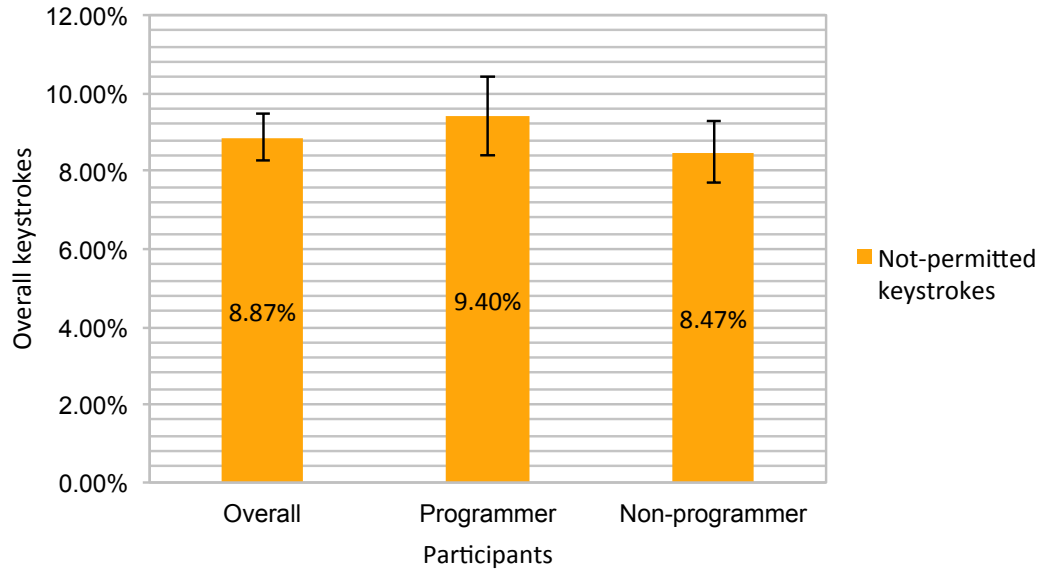


Figure 6.11. Percentage of all the keystrokes classified as not-permitted.

Overall		Programmers		Non-programmers	
Delete	(80%)	Delete	(79%)	Delete	(80%)
Enter	(13%)	Enter	(13%)	Enter	(13%)
,	(2%)	,	(3%)	,	(1%)
\	(1%)	\	(1%)	.	(1%)

Figure 6.12. Top 4 not-permitted keys with their frequency (in percentage).

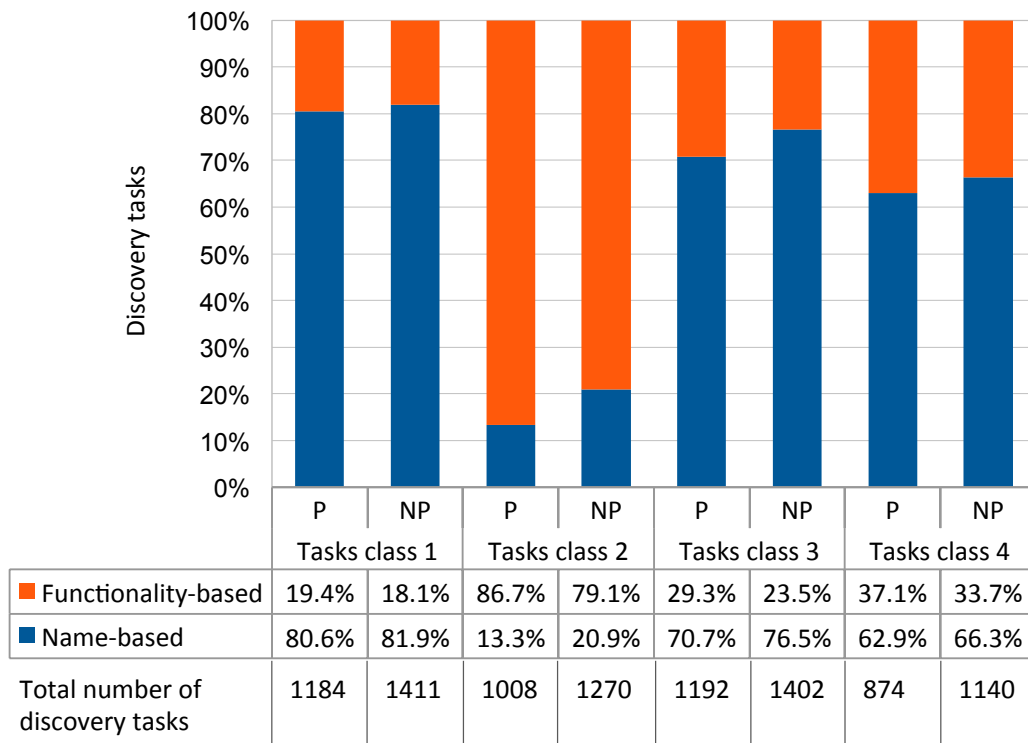


Figure 6.13. Ratio of using the two discovery methods segmented by the participant groups and task classes.

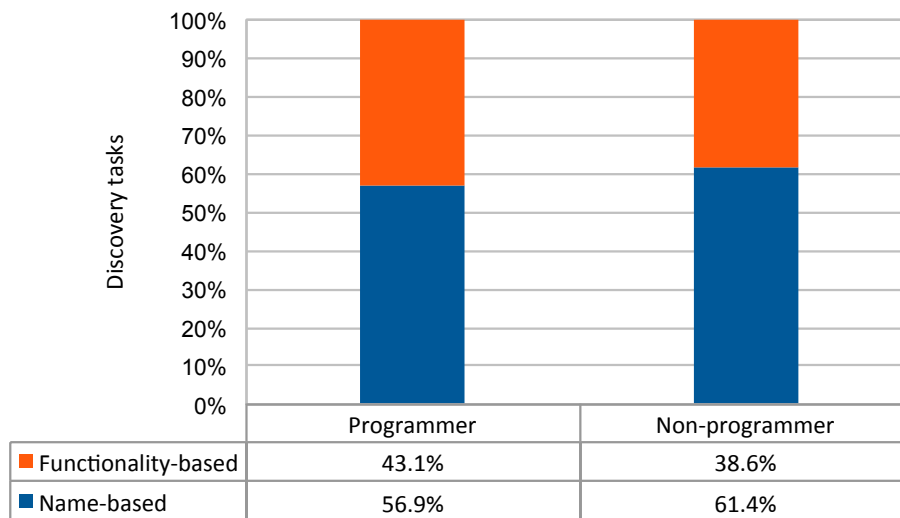


Figure 6.14. Difference between programmers and non-programmers in terms of the ratio of using the discovery methods.

6.7 Conclusion and Lessons learned

In summary, the results answered all the evaluation questions, approved our hypotheses, and more importantly showed that *NaturalMash* is usable by non-programmers. To be specific, the accuracy of 82% and the low average completion time of less than 3 minutes for the pre-constructed tasks, as well as the level of perceived ease-of-use for the main features of the system (3 out of 4) all prove the practical usability of the system.

Since programmers and non-programmers both obtained a high level of efficiency, the answer to **RQ1** is that the system is highly usable for end-users with different programming backgrounds. The equivalence test results on the efficiency of programmers and non-programmers showed that the hypothesis **H1** is also true (i.e., having a programming background does not have significant impact on the usability of the system).

On the other hand, the results provided evidence that there is a difference between programmers and non-programmers in terms of favoring either of the API discovery methods, and that **H2** is true. As an answer to **RQ2**, we can thus say that programmers prefer the text field over the ingredient toolbar, whereas non-programmers prefer it the other way around. On the other hand, for API discovery tasks, the ingredient toolbar is perceived by both the groups to have a higher level of ease-of-use than the text field. A possible explanation is the difference between the mental model of programmers and non-programmers. Programmers are used to text-based environments, while non-programmers do not have this bias and naturally use the method they perceive to be easier to use. Moreover, the higher level of perceived ease-of-use for the ingredient toolbar might be rooted in the accuracy of the name-based discovery method, as opposed to the functionality-based discovery method that, depending on the ambiguity of the input keywords, is probable to produce irrelevant suggestions in the text field. In fact, the rationale behind introducing the functionality-based discovery method was to let end-users find their desired ingredients when they are not able to directly recognize and target them using the name-based method. Our rationale was approved by the higher rate of using the functionality-based discovery method (by both programmers and non-programmers) in the tasks of class 2, where the list of to-be-used ingredients are not given.

The usefulness of the system is supported by the high SUS score (82) as well as the positive answers to the open-ended and self-reported questions on the perceived usefulness of the system. However, the low accuracy in the open task may seem to hinder proving the usefulness of the system. As it was indicated in the results, the main reason for the high amount of “skipped” and “incorrect”

tasks was lack of required ingredients. Even though it is clearly a limitation perceived by the participants, — this approves **H3** — it is not by definition within the scope of the core design, as the library can always be extended by new ingredients (no matter what technology they use). Therefore, the answer to **RQ3** is that the system is potentially useful in real situational use cases, provided that the needed ingredients are available in the library.

The evaluation also helped us identify a number of usability problems that we intend to address with the following solutions:

- **Stateful live programming.** The low perceived usefulness of the live output compared with the rest of the features is caused by a usability problem pointed out in some of the participants feedback (e.g., comments left while doing the tasks, and answers to the exit open-ended question in the exit questionnaire). For instance:

“The refresh frequency is a bit annoying. Every time I type even a single character, something is moving, refreshing, or updating.”

Every time a single change is made in the text field, the visual field gets updated by re-compiling and re-executing the output mashup. However, the way this mechanism is currently implemented, i.e. the new mashup replaces the old one in the output Iframe, introduces two major usability problems. First, during the replacement of the output mashup the visual field may flash multiple times. This can cause distraction and interrupt the flow of the user’s activity. Second, when the old mashup is replaced, the state of its user interface will be vanished too as though it is reset. The end-user will then have to manually alter the user interface state to what it used to be. Using model-driven engineering, we can store the old state and automatically restore it in the new mashup user interface. To avoid flashing during the replacement and state restoration processes, one solution is to display the new mashup only when it is completely loaded.

- **Crowdsourced ingredients development** We verified in this evaluation that the main perceived limitation of the system is revealed when the required ingredients are missing. As a matter of fact, there is a large and rapidly growing number of Web APIs, providing various services, contents, and data. One practical solution to keep the ingredient library up-to-date is therefore to use crowdsourcing. In doing so, non-programmers in need of certain ingredients can make a call to the crowd (i.e., other end-users of the system) with all the relevant information. This information is partially generated by the system (e.g., the keywords used in the failed discovery, and the mashup recipe written right before the failed discovery task), and then completed with the explanation of the

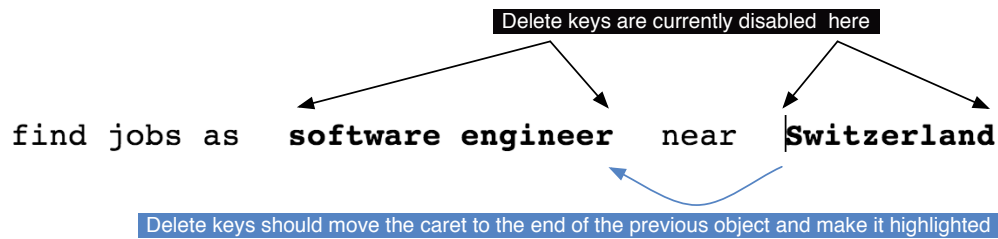


Figure 6.15. The text field should lift the restriction on using Delete keys at the beginning and end of data objects.

end-user about the required ingredient. Professional end-users (e.g., programmers), who we call ingredient developers and are familiar with Web APIs, will respond to the call by finding, developing, and adding the requested ingredients to the library. Like any other crowdsourcing approach, there can be some sort of incentives (e.g., money) to motivate the ingredient developers. Moreover, non-programmers should be able to choose to whether share their new ingredients publicly, with a certain end-users, or even privately.

- **Enhanced deletion in the text field** The low percentage of using not-permitted keys (around 9%) to a large extent shows that we have applied the right restrictions in the semi-structured text field. On the other hand, the fact that Delete keys were the most frequently used not-permitted keys (around 80%) may indicate a potential usability problem. In the text field, Delete keys are not permitted at the beginning and end of data objects as though the text of the object is isolated from the rest of the recipe. The fact that all the non-permitted Delete keys were used in these areas (beginnings and ends of objects) suggests that the isolation of objects is not an appropriate restriction. Therefore, the text field should allow end-users to continue deleting the previous or next object when they reach the beginning or end of the object they are deleting (Figure 6.15). In doing so, highlighting or blinking can be used to notify the end-users that they are about to delete (or edit) a new object.

Chapter 7

Comparative Evaluation of Mashup Tools

One of the major criticisms of the current research in mashups is that the majority of the proposed mashup tools in literature do not come with adequate and appropriate evidence on their superiority over existing ones. As a result, research in mashups has not yet reached its full potential. In addition to its scientific importance, evaluation of existing mashup tools gives valuable insight and guidance for both industrial mashup tool designers as well as end-users who are to select a proper tool suiting their needs.

An impediment to such an evaluation is the lack of agreed benchmarks for how well a mashup tool is designed. In this case, however, benchmarking can become a very challenging task. First, there are more than one aspect against which a tool can be benchmarked. Some of these aspects have been considered previously but not thoroughly. For instance, [34] proposes a framework which assesses mashup tools against their expressive power to create mashups. In terms of the composition environment, [48] collects and categorizes some of the features offered by current mashup tools into a coherent framework. Second, mashup tools have been evolving rapidly in terms of architecture and capabilities [1]. This makes it significantly difficult to arrive at a set of common measurement indicators. Likewise, due to their focus on end-user development it may be difficult to rely exclusively on quantitative metrics as some qualitative aspects of the tool usability may need to be taken into account in the evaluation.

In this chapter, we propose a benchmarking framework for mashup tools (Section 7.1). Using the framework, we also comparatively evaluate the state-of-the-art tools (including *NaturalMash*) against their expressive power (Section 7.2).

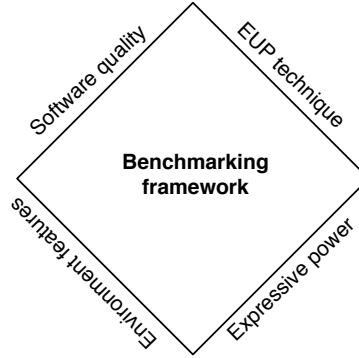


Figure 7.1. A benchmarking framework for mashup tools with four dimensions.

7.1 A Benchmarking Framework for Mashup Tools

In the following, we organize our proposed benchmarking framework into four dimensions (Figure 7.1), namely *EUP techniques*, *environment support features*, *expressive power*, and *software quality*. We explain these dimension, and within each dimension, in turn, we discuss a number of possible measurement indicators. Finally, we explain the strategy that should be adopted to use the proposed benchmarking framework.

Dimension: EUP Techniques

As discussed in Chapter 2, Mashup tools use various EUP techniques to facilitate the development of mashups. Each technique provides a different level of abstraction and expressive power. Thereby, mashup tools can be benchmarked against the techniques they accommodated in their design. Since they usually combine two or more techniques, it may be necessary to select the major techniques of interest and only subject that one to the benchmark. For example, in case of *Yahoo! Pipes* the major technique is visual language programming, even though it also utilizes form-based technique. The challenge is, however, to arrive at a set of factors for benchmarking EUP techniques. These factors should focus on the design of the interaction technique, and thus should not be confused with usability evaluation. For instance, [24] proposes a design benchmark for visual programming languages, which appear to be widely utilized by existing mashup tools. The proposed benchmarking framework contains various indicators extracted from the generic characteristics of visual programming languages such as multidimensionality (e.g., 2D vs. 3D visual languages).

Dimension: Environment Support Features

The composition environment of a mashup tool is of utmost importance, as this is where the end-users accomplish complex development tasks. As a matter of fact, the composition environment should provide adequate features supporting and guiding end-users through their tasks. These features may include collaborative development [46], online community [48], liveness [1], suggestion mechanism [35], and so on. Evaluation within this dimension can be performed either horizontally or vertically. In the former case, the evaluation can be based on the number of features each tool can offer. In doing so, each feature might have a different weight to the final sum. The same features might be implemented by different tools using different methods. The latter form of evaluation is applicable when the methods are the subject of evaluation. For instance, in case of recommendation mechanism, one may be interested in evaluating the efficiency or performance of existing recommendation mechanisms.

Dimension: Expressive Power

The expressive power of a mashup tool represents how many different classes of a mashup can be generated using the tool. In terms of model-driven engineering, the expressive power of a tool stems from its composition and component model. The more heterogeneous types of Web APIs a component model can describe, the more expressive it is. Likewise, a composition model describes different ways the Web APIs can be glued together, and the more ways of composition the model supports, the higher level of expressiveness it possesses.

Dimension: Software Quality

Depending on the domain where the mashup tool is going to be deployed, different quality requirement might be imposed on the design of the tool. These quality aspects concern not only the mashup tool as whole, but also the mashups generated by the tool.

- **Mashup Tool:** Some of the most easy-to-measure quality attributes that a mashup tool might be benchmarked against are performance (in terms of the responsiveness of its user interface or the overhead of the mashup execution runtime) and scalability (in terms of the number of components and mashups models that can be managed at design-time and executed concurrently at runtime). The usability of the mashup tool is also an important quality to evaluate, which may be difficult to measure without recurring to end-user studies. This

quality is also correlated with the choice of interaction techniques and the features of the composition environment of the tool, which have been previously mentioned.

- **Generated Mashup:** In addition to its runtime performance, one of the key quality attribute for the generated mashups by a tool concerns its security. Since mashups expose potential security threats, as they combine third-party data from untrusted, anonymous sources on the Web [80].

7.1.1 Benchmarking Strategy

Benchmarking for mashup tools is based on the same rationale and methodology as in software engineering and performance optimization. In the following, we discuss a possible benchmarking strategy tailored for mashup tools.

- **Select appropriate benchmarking aspects for comparison.** The first step is to have a clear goal for the benchmarking process. It helps to narrow down the evaluation process by selecting only the necessary benchmarking aspects from the whole framework.

- **Know existing mashup tools.** Before starting the benchmarking process, it is necessary to have sufficient knowledge about existing mashup tools. To this end, a comprehensive survey, that is currently lacking in literature, seems essential. A mashup tool should not be necessarily compared against all the existing tools, but a subset of them, determined by the goal of the benchmarking process and how the mashup tool can be classified.

- **Use or read about the mashup tools under evaluation.** After determining the list of mashup tools subject for evaluation, it is important to learn about them through either using them (Web-based mashup tools should be readily accessible online) or reading about them (if they are described in academic publications but lack an online release).

- **Understand and bridge the gap.** If the goal of the benchmarking is to evaluate a mashup tool with respect to a specific aspect, the results inform either the success or failure of the design. In the later case, it is important to analyze the results to understand how to overcome the weaknesses.

7.2 Benchmarking the State-of-the-art Mashup Tools

In this section, we use the framework to compare *NaturalMash* with the state-of-the-art tools in terms of expressive power. We show that our system offers a moderate and competitive level of expressive power.

Mashup Tools	Data Integration		Process Integration				Presentation	
	Aggregate	Filter	Sequence	Condition	Loop	Event	Layout design	Wiring
NaturalMash	×	×	×	–	–	×	×	×
Mashroom [112]	×	×	×	–	–	–	×	–
Husky	×	×	×	–	–	–	–	–
Karma [109]	×	×	×	–	–	–	–	–
MashMaker [38]	×	×	×	–	–	–	×	×
Vegemite [75]	×	×	×	–	–	–	–	–
Yahoo! Pipes	×	×	×	–	×	–	–	–
IBM Mashup Center	×	×	×	–	×	–	×	×
JOpera [94]	×	×	×	×	×	×	×	–
JackBe Presto	×	×	×	–	×	–	×	×
Marmite [114]	×	×	×	–	–	–	–	–
MashArt [33]	×	×	×	–	–	–	×	×
ResEval Mash [64]	×	×	×	–	–	–	–	–
MyCocktail	×	×	×	–	×	–	×	×
MashableLogic	×	×	×	–	–	–	×	×
Swashup [83]	×	×	×	×	×	×	×	×
WMSL [100]	×	×	×	×	×	×	×	×
ServFace [90]	×	×	–	–	–	–	×	×
DashMash [25]	×	×	–	–	–	–	×	×
Omelette [28]	×	×	–	–	–	–	×	×
CRUISE [96]	×	×	–	–	–	–	×	×
RoofTop [61]	×	×	–	–	–	–	×	×
d.mix [55]	×	×	×	×	×	–	×	×
Open Mashup [18]	–	–	×	–	–	–	–	–
IFTTT	–	–	×	–	–	×	–	–
SABRE	×	×	×	×	–	×	×	–
MashStudio	×	×	×	×	–	×	×	–

Table 7.1. Comparison of the state-of-the-art mashup tools in terms of **composition model** expressive power. The light gray lines highlight mashup tools with more composition expressive power than NaturalMash (highlighted with a dark gray line).

Mashup Tools	Data Formats			Access Methods			Output			Behavior	
	Primitive	Complex	User-defined	Language	Protocol	Database	Data	Logic	Presentation	Task	Event
NaturalMash	×	×	×	×	×	×	×	×	×	×	×
Mashroom [112]	×	×	-	-	×	-	×	-	-	×	-
Husky	×	×	-	-	×	-	×	×	-	×	-
Karma [109]	×	×	-	-	×	×	×	-	-	×	-
MashMaker [38]	×	×	-	-	×	-	×	-	-	×	-
Vegemite [75]	×	×	-	-	-	-	×	-	-	×	-
Yahoo! Pipes	×	×	-	-	×	-	×	-	-	×	-
IBM Mashup Center	×	×	-	-	×	×	×	×	×	×	-
JOpera [94]	×	×	×	×	×	×	×	×	×	×	×
JackBe Presto	×	×	×	×	×	×	×	×	×	×	×
Marmite [114]	×	×	-	-	×	-	×	×	-	×	-
MashArt [33]	×	×	-	-	×	-	×	×	-	×	-
ResEval Mash [64]	×	×	-	-	×	×	×	×	-	×	-
MyCocktail	×	×	-	-	×	×	×	×	-	×	-
MashableLogic	×	×	-	-	×	×	×	×	×	×	×
Swashup [83]	×	×	-	×	×	×	×	×	×	×	×
WMSL [100]	×	×	-	×	×	×	×	×	×	×	×
ServFace [90]	×	×	-	-	×	×	×	-	×	×	×
DashMash [25]	×	×	-	-	×	-	×	-	×	×	×
Omelette [28]	×	×	-	-	×	-	×	-	×	×	×
CRUISE [96]	×	×	-	-	×	-	×	-	×	×	×
RoofTop [61]	×	×	-	-	×	-	×	-	×	×	×
d.mix [55]	×	×	-	×	×	-	×	×	×	×	×
Open Mashup [18]	×	-	-	-	×	-	×	×	-	×	-
IFTTT	×	×	-	-	×	-	×	×	-	×	-
SABRE	×	×	-	-	×	-	×	×	-	×	-
MashStudio	×	×	-	-	×	-	×	×	-	×	-

Table 7.2. Comparison of the state-of-the-art mashup tools in terms of **component model** expressive power. The light gray lines highlight mashup tools with (almost) the same component model expressive power as NaturalMash (highlighted with a dark gray line).

7.2.1 Methods

We reused the list of the mashup tools surveyed in Chapter 2, and added *NaturalMash* to the list, which resulted in a total of 28 mashup tools. We followed the benchmarking framework presented in this chapter. In doing so, we picked the expressive power dimension, which, among other dimensions, is the most relevant to this dissertation.

As mentioned in the benchmarking framework, the overall expressive power of a mashup tool is characterized by the expressive power of its composition and component models. Therefore, separately for component model and composition model, we proposed a number of factors, in the form of a checklist, that together define the expressive power of the model. We attempted to include as many factors as possible that have a sense of generality and can be assessed for all mashup tools in the list.

Assuming all the factors have the same weight, in order to actually measure the expressive power of a tool, we simply summed up the number of factors (both in composition and component model) supported by the tool — in other words, each factor is a binary metric.

Composition Model

In general, mashup tools can be composed at three layers: *data integration*, *process integration*, and *presentation*. [54]. Within each layer, thus, different mashup tools provide different levels of expressive power.

- **Data Integration.** Data integration is accomplished by aggregating and filtering data sources [37]. Data aggregation is the process of transforming scattered data from numerous sources into a single new one. Data filtering is the process of specifying conditions on a data source to extract the exact information being sought.

- **Presentation Integration.** Process integration involves constructing the application logic using *conditions*, *loops*, and *sequences* as well as catching *events* triggered by Web APIs (e.g., receiving updates in a stream of data). The difference between conditions and events is that the latter is triggered by Web APIs, whereas the former is implemented internally within a mashup tool.

- **Presentation Integration.** At the presentation layer, various widgets can be superficially rearranged (*layout design*) or *wired* with each other. Wiring is the process of connecting to widgets in which one end of the wire represents an event fired by one of the widgets, and the other end is attached to a functionality offered by the other widget.

Component Model

The factors that affect the expressive power of a component model, are also the aspects that technically characterize Web APIs. A Web API is technically specified by the *data formats* used by its input/output parameters, *access methods* that should be utilized to remotely communicate with it, *application layer* it contributes to, and control flow *behavior* it carries inside a mashup.

- **Data Formats.** Within a mashup, a Web API interacts with others through its input and output parameters. To make this interaction possible, the data formats used by these parameters should be defined in the component model, so that the tool can undertake (or suggest users to perform) necessary data mapping and matching processes. In general, these data formats can be *primitive*, *complex*, or *user-defined*. Primitive formats are equivalent to standard variable types of a programming language (e.g., string, int, boolean, etc.). On the Web, complex formats usually constitute MIME types (e.g., XML, JSON, and RSS). Additionally, many data formats are arbitrary (not standard) and classified as user-defined.

- **Access Methods.** Access method is the way a Web API is made accessible for composition inside a mashup. The access methods utilized by various Web APIs are highly heterogeneous. Some Web APIs force the use of a specific programming, scripting or markup *language*. For instance, JavaScript APIs, HTML IFrame widgets, Plain Old Object Java Objects (POJOs), Enterprise Java Beans (EJB) can be all considered within this category. Even though some of these methods are considered outdated (POJO and EJB), they are still being used within enterprises. Moreover, Google Maps API, which is the most popular mashup components according to the ProgrammableWeb, is mainly accessible via JavaScript APIs. However, Web APIs mostly utilize standard *protocols* such as Web services (e.g., RESTful, HTTP and SOAP) and Web feeds (e.g., RSS and Atom). According to the ProgrammableWeb, the dominant portion of Web APIs currently utilize REST Web services. Moreover, within a (enterprise) mashup, a *database* can be encapsulated as a Web API and act as either a read-only or a read/write data source. A database not only can deliver data and functionality (i.e., query and update features) but also can become a permanent storage for writing user-related data (e.g., username and password).

- **Layer.** There are three forms of output a Web API can produce in the final mashup composition: *data*, *application logic*, and *presentation*. As discussed above in the composition model, the development of a mashup can span one or all of the integration layers. However, supporting a specific layer form by a component model does not necessarily imply the feasibility of composition at

that layer. For instance, *JOpera* accommodates a component model that also supports presentation layer, while it does not allow composition at that layer. Web APIs generating data act as external data sources, which deliver data to a mashup either as continuous data streams with real-time properties or as snapshots of a remote or local dataset. Most Web data sources are read-only, but in some cases they may also support updates. Within the mashup, they are likely to be converted, transformed, filtered, or combined with other data sources [82]. Web APIs with logic output are delivered as services that contribute to the business logic layer of a mashup. Such Web APIs are usually orchestrated together in a workflow to deliver a capability [36]. Visual output is generated by widgets [60]. Widgets provide some kind of graphical user interaction mechanism which can be reused at the mashup user interface level. The visual part of a widget is incorporated in the mashup user interface independently from other widgets. Widgets may also generate reusable logic and data accessible for composition within a mashup (e.g., Google Maps API).

- **Behavior.** At the runtime, the control flow of a mashup determines the sequence of component invocation. Nevertheless, the internal execution mechanism of a Web API may also affect its parent mashup control flow. This is referred to as the runtime behavior of a Web API that can be either *task-based* or *event-based*. A task-based behavior represents a single invocation of a local or remote operation, which may provide an output given an input. It resembles traditional functions or methods, which execute and transmit responses only when called. In the context of the overall mashup, task-based Web APIs are passive (they are executed only when control reaches them). When a Web API has an event-based behavior, it is triggered and produces an output only when a specific action (independent from the composition) has been taken (e.g., user interactions or an asynchronous message is received from a remote service). An event-based Web API is, therefore, an active part of a mashup, which may trigger the execution of a sequence of tasks.

7.2.2 Results and Discussion

As it can be seen in Tables 7.1 and 7.2, industrial mashup tools (e.g., *IBM Mashup Center*, and *JackBe Presto*) generally offer much more expressive power than academic tools. Interestingly, looking into the results, it can be seen that they conform to the expressive power dimension (low, moderate, and high) of the descriptive classification we conducted in Chapter 2, Section 2.5.

In terms of composition model, only a few mashup tools including *Swashup*, *WMSL*, *JOpera*, and *d.mix* offer more expressive power. Most of these tools are

based on textual DSLs, which are tailored for the development of mashups, and reasonably offer a high level of expressiveness. *NaturalMash*, however, lacks the support for expressing loops and conditions, which are supported by non-DSL tools like *Yahoo! Pipes*. We believe, explicitly supporting these control flow constructs is not very much useful. In fact, iterating over data, for instance, can be automated without users having to explicitly use loops to do so (we plan to implement this in future). Moreover, supporting data extraction inside *NaturalMash* seems essential and can be done through incorporating a third-party mashup enabler like *Dapper*.

Concerning the component model, *NaturalMash* fully meets all the factors, thanks to its expressive component model (Chapter 3, Section 3.3). Other tools with slightly less expressive component model include *Swashup*, *WMSL*, *JOpera*, and *JackBe Presto*. The ability to handle user-defined data formats, despite its importance, is rarely supported by the reviewed component models. As described previously (Chapter 3, Section 3.3), *NaturalMash* provides a convenient way to create and use user-defined data formats. Another advantages of the *NaturalMash* component model is the support for language-based protocols (e.g, widgets with JavaScript SDK), which is not however supported by most of the models. Another overlooked protocol by existing models is database, which is of value in the enterprise market [60], even though it is not commonly used in the so-called consumer mashups [72].

Overall, the results show that *NaturalMash* offers a moderate and competitive level of expressive power that is less than DSL-based tools like *Swashup* (highly expressive tools), and more than WYSIWYG tools such as *DashMash* (highly usable by low expressive tools).

Chapter 8

Conclusion

With the emergence of the long-tail in the market of software applications, computer programming is no longer viewed as a task for just programmers. The consequence was the rise of the so-called citizen developer: ordinary end-users developing niche software applications within specific domains. These niche applications are not limited to spreadsheets and databases (as it used to be). Rather, they are small-scale enterprise applications addressing unique, personal, and transient needs of end-users. One popular form of such applications is called mashup, built on the Web out of a lightweight composition of Web APIs (reusable software components delivered as a service through the Web). Thanks to the reuse of Web APIs, the development of mashups demands lower costs in terms of time and required skills. In return, they offer a high level of added value to the end-users, thanks to the diversity and ever-growing number of Web APIs that can be used as their building blocks.

Nonetheless, the vision of citizen developer is problematic: Can end-users (i.e., citizen developers) build their own business application? If not, how they can be supported to do so? These problems arise from the end-users' lack of knowledge and will to learn programming or scripting languages required to create any type of software, including mashups. Hence, end-users cannot really build their own applications unless they are supported. End-User Development (EUD) is an emerging research field dealing with this type of problems. It is geared towards developing tools and techniques facilitating the development of software applications by non-programmers.

In this dissertation, we designed and implemented an EUD system that empowers end-users to develop mashups as niche software applications. EUD systems for mashups is commonly referred to as mashup tools. We assumed that mashup tools confront a trade-off between expressiveness and usability for non-

programmers, and that the sweet-spot in this trade-off can be hit if the tool is able to provide a proper combination of EUP techniques ensuring both high level abstraction and expressiveness, metaphors familiar to everyone, and an optimal learning experience developing the skills of non-programmers. Within this assumption, we hypothesized that such a mashup tool can adopt a hybrid EUP technique combining natural language programming and WYSIWYG, the “cooking” metaphor in which the “ingredients” (Web APIs) are mixed according to a “recipe” (natural-language source code) for the purpose of preparing “food” (mashup), and live programming and recommendation techniques to boost optimal learning experience.

Next, we surveyed the state-of-the-art mashup tools based on a decision space concerning their design. Given the implications of the survey results, we roughly classified the reviewed tools based on their expected level of expressiveness and usability for non-programmers, two important qualities that should be taken into account in the design of any mashup tool. Within this classification, we characterized our hypothesized system as an optimal usable mashup tool that provides a moderate level of expressiveness, while it ensures the highest level of usability for non-programmers.

We thoroughly presented the design and implementation of *NaturalMash*, as our hypothesized optimal usable mashup tool. *NaturalMash* combines natural language programming with WYSIWYG and Programming by Demonstration in a simple single-page user interface. The architecture of the system is geared towards enabling the user’s experience of live programming by ensuring high performance, minimizing the edit/compile/execute response time. Furthermore, the design of the system adopted a user-centered design process taken over a two-year period. We explained the design process by reporting the formative evaluations conducted in three iterations. In the first iteration we performed an expert review with an early prototype of the system in order to find early usability problems. The second and third iterations involved usability testing undertaken with samples of participants, diverse in terms of programming knowledge and skills. In reporting the evaluations, we also showed how the results drove the design of *NaturalMash* during the process.

Finally, we validated our hypothesis by evaluating *NaturalMash*. The summative evaluation was conducted using a usability testing on a sample of 40 participants with diverse programming background. The results suggested that the system is usable by non-programmers. To conduct the comparative evaluation, we first proposed a benchmarking framework providing a general picture on possible methods to actually measure mashup tools against each other, and used the framework to compare *NaturalMash* with the state-of-the-art tools against

their expressiveness level. The results from both summative and comparative evaluations suggested that *NaturalMash* is highly usable by non-programmers, and offers a moderate yet competitive level of expressive power, which together indicate that the system could be considered as being our hypothesized optimal usable mashup tool.

To conclude, in this dissertation we addressed a key problem in the design of EUD systems, being that how to maximize the expressiveness without compromising the usability for non-programmers. We believe our approach, summarized below, can be reused in the design of next-generation EUD systems.

The design of mashup tools confronts a major trade-off between expressiveness and usability for non-programmers, that should be fine-tuned to achieve the desired design. Given the problem of supporting citizen developers to exploit the market of niche software applications, we believe the appropriate balance in the trade-off is achieved by maximizing the level of usability for non-programmers, and increasing the expressiveness to reach a sufficient level. To this end, the most important design decisions concern choosing and combining EUP techniques with the right level of abstraction and expressiveness (e.g., natural language programming combined with WYSIWYG and PbD), using familiar metaphors for mashup development (e.g., the “cooking” metaphor), and providing an optimal learning experience to support and develop end-users’ abilities (e.g., through enabling live programming, and providing recommendations and suggestions).

8.1 Limitations

The research presented in this dissertation has a number of limitations which are described subsequently.

8.1.1 Comparative Usability Evaluation

We did not conduct a comparative usability evaluation to see how usable our system is compared with the state-of-the-art mashup tools. We argue that performing such an evaluation is implausible, and beyond our resources. First, in order to conduct a comparative usability evaluation, we had to get our hands on all the mashup tools, which might not be feasible due to the fact that the major-

ity of them are lab-based research prototypes. Second, due to the heterogeneity and large number of mashup tools, it would be extremely challenging to derive a set of common tasks that can be accomplished by all the mashup tools. Even if we could, we most probably would end up with a list of tasks that are too simplified to be of practical use in comparing the usability of the tools. Hence, our comparative evaluation was focused on the expressive power.

8.1.2 Summative Usability Evaluation

The summative evaluation presented in Chapter 6 has two main limitations. The first concerns the imbalanced number of participants recruited using different methods. Participants recruited through crowdsourcing were plausibly motivated by the incentive, whereas the invited participants were only interested in trying out the system. We, therefore, attempted to have an approximately equal number of invited and crowdsourced participants. This, however, did not work out and we managed to recruit only 10 participants by invitation. The reason might be due to the fact that the interest of the invited people in trying out the system was not enough to persuade them to finish the study. After inspecting the logs, we realized that the invited participants, who failed to complete the study, had been stuck in the first task, and had spent a lot of time on just trialling the system.

The second limitations is that the open task is self-generated after conducting the pre-constructed tasks. This introduced a bias causing the similarity between some of the self-generated tasks and the pre-constructed tasks. Furthermore, we wanted the participants to freely come up with a personal use-case without considering and knowing about the system limitations (i.e., getting biased by the pre-constructed tasks), as the goal here was to measure how useful *NaturalMash* is in real situational use-cases. However, we were forced to admit these biases and postpone self-generating the open task to the end due to one main reason: According to our experience with past formative evaluations, we anticipated that most of the participants are not familiar with mashups and Web APIs and that the tutorial alone might not be enough to introduce these concepts properly. Therefore, we predicted that many participants will fail to self-generate a task in the beginning of the study.

8.1.3 Comparative Expressive Power Evaluation

In Chapter 7, we proposed a benchmarking framework, and according to that, conducted a comparative evaluation of the state-of-the-art mashup tools against

their expressive power. A mashup tool usually comes with a library in which the end-users can find the required Web APIs. The more the number of available Web APIs within a tool, the higher its level of expressive power. Also, as it was reported by the participants in the summative evaluation (Chapter 6), the main limitation of *NaturalMash* was the lack of required Web APIs, suggesting that it also influences the perceived expressive power of a tool (beside the “real” expressive power measured in this chapter). In the comparative evaluation, however, we excluded this factor because many of the surveyed mashup tools are in fact research prototype, to which we did not have real access. Also, acquiring the required information from the authors of these tools was not an option, because, again, a prototype comes with a limited list of Web APIs, which would be unfair to be compared with industrial tools.

8.2 Future Directions

Beside implementing the solutions to the usability problems we found in the summative evaluation (see Chapter 6, Section 6.7), we envision to conduct future research in the following directions:

- **End-user development of physical mashups.** The Internet of Things is a paradigm for the future Internet in which everyday objects are equipped with sensors and connected to the Internet. One of the emerging visions [15] correlated with this paradigm is the so called Web of Things [50]. The goal of this vision is to use Web standards and technologies to facilitate the connection and integration of everyday objects. Once these objects are exposed and accessible as Web APIs, they can be composed with other Web APIs that may or may not encapsulate physical objects. The resulting composition applications are called physical mashups [51] and provide added value across the Internet of Things. Physical mashups are similar to traditional Web mashups with the difference that they also compose physical Web APIs (i.e., Web APIs encapsulating physical objects). For example, a physical mashup can give insight into the efficiency of a home heating system by displaying weather information (e.g., from a weather Web API) alongside data from different sensors connected to the heating system. The development of physical mashups follows a long tail model, in which countless situational and unique needs of users can drive the development of mashups. Therefore, EUD can be applied to enable the development of physical mashups in a “do-it-yourself” manner. So far, existing relevant studies have mainly focused on the engineering challenges, such as developing a unified interface for physical Web APIs [105] and designing a scalable architecture for the Web of Things [52].

We will build on these works and extend them if necessary. However, identifying and tackling the problem of self-service physical mashup development through an HCI perspective has previously received little research attention.

- **Enterprise data integration using natural language programming.** In the enterprise integration industry (e.g., Crossing Tech¹), multiple actors such as business analysts, architects and developers are inevitably involved in the process of building an integration solution. These actors represent two groups known as functionals and technicals. There is a clear segregation between these two groups in terms of communication as they clearly don't have the same expertise, don't share the same goals and mostly don't use the same terminologies. At the beginning of an integration project, the business analysts usually describe in a text document what they expect the solution to do and how the different entities are going to be mediated. This business documentation is to be given to and analyzed by the architect to design an integration architecture. A new set of documentation artifacts are created by the architect to describe in more details the diagram, the architecture and its components. This architecture documentation will then be provided to the developers who will implement the components. Nevertheless, the original business documentation is never delivered to the developers. This situation often leads to misunderstandings as the communication gap is a serious issue between the business analysts and the developers.

This gap can be addressed through the lens of EUD. To be specific, we plan to study how to use a controlled natural language to model complex enterprise integration solutions, which are currently visualized using technical diagrams that seem to be difficult to understand for business users. The advantage of using an executable controlled natural language is that it is not only understandable by business users, but also as expressive as visual diagrams.

¹<http://www.crossing-tech.com/>

Appendix A

Usability Evaluation Forms

This appendix contains questionnaire forms used in the usability evaluations of *NaturalMash*: formative usability evaluation of iteration 2 (Chapter 5, Section 5.2), formative evaluation of iteration 3 (Chapter 5, Section 5.3), and summative evaluation (Chapter 6).

A.1 Background Assessment Questionnaire

1. **How often, if ever, do you use social networks (e.g., Facebook, Twitter, LinkedIn, etc.)?**

Never 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Very frequently

2. **How often, if ever, do you check Web feeds, blogs, and podcasts?**

Never 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Very frequently

3. **Do you have a Website or blog?**

☐ Yes

☐ No

4. **Which of the following services have you used?**

☐ Flickr

☐ Facebook

☐ Twitter

☐ LinkedIn

☐ Last.fm

- ☐ eBay
- ☐ Google Maps
- ☐ Google Search
- ☐ Google News
- ☐ Delicious
- ☐ Eventful
- ☐ BBC News
- ☐ SlideShare

5. **How familiar are you with spreadsheet programs (e.g., Microsoft Excel)?**

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

6. **How familiar are you with editing wikis? (e.g., editing Wikipedia)?**

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

7. **How familiar are you with Website builders? (e.g., WordPress, Wix, Weebly, etc.)?**

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

8. **How familiar are you with HTML?**

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

9. **How familiar are you with XML?**

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

10. **How familiar are you with JavaScript?**

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

11. How familiar are you with Linux shell script?

No familiarity 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Expert

12. Please, write down the list of programming languages you speak.

13. Please indicate years of continuous experience in programming (the learning period is also counted).

- ☐ 0-1 year
- ☐ 1-2 years
- ☐ 2-5 years
- ☐ 5-10 years
- ☐ 10+ years

14. Please indicate years of experience in industry (i.e., working as a developer in a company).

- ☐ 0-1 year
- ☐ 1-2 years
- ☐ 2-5 years
- ☐ 5-10 years
- ☐ 10+ years

A.2 Exit Questionnaire: First Iteration Formative Evaluation

1. How you felt about the system (please use a scale from 1 to 5)?

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

Frustrating 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Satisfying

Dull 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Stimulating

Rigid 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Flexible

2. The autocompletion list simplifies the editing task for adding new labels:

Strongly disagree 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Strongly agree

3. Selecting appropriate labels from the autocompletion menu:

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

4. Highlighting the text:

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

5. Exploring new labels by trial and error:

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

6. Autocompletion list helps memorizing the labels:

Strongly disagree 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Strongly agree

7. Correcting your mistakes:

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

8. Task 1:

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

9. Task 2:

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

10. Task 3:Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**11. Task 4:**Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**12. Task 5:**Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**13. Do you want to continue using the tool?**☐ Yes☐ Maybe☐ No**14. Please tell us briefly why?**

A.3 Exit Questionnaire: Second Iteration Formative Evaluation

1. How you felt about the system (please use a scale from 1 to 5)?

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

Frustrating 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Satisfying

Dull 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Stimulating

Rigid 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Flexible

2. Autocomplete menu (the menu that appears as you type in the text field):

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

3. Label suggestions provided by the autocomplete list (the list that appears as you type in the text field):

Irrelevant 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Relevant

4. Correcting your mistakes in the text field:

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

5. Being able to see the output updated as you type:

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

6. Visual editor suggestions (getting suggestions in the text field while interacting with the output):

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

7. Interacting with the output:

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

8. Highlighting (highlights for texts, Icons, and windows):

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

9. Ingredients toolbar (the toolbar that contains the list of ingredients):Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful**10. Task 1:**Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**11. Task 2:**Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**12. Task 3:**Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**13. Task 4:**Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy**14. Do you want to continue using the tool?**☐ Yes☐ Maybe☐ No**15. Do you want to suggest the tool to your friends?**☐ Yes☐ Maybe☐ No**16. Please tell us briefly why?**

A.4 Exit Questionnaire: Summative Evaluation

1. **Do you have any personal use case that can be potentially addressed using NaturalMash?**

2. **Do you have any additional information or comments to share?**

A.5 Post-task Class 1 Questionnaire: Summative Evaluation

1. **Label suggestions provided by the autocomplete menu (the menu that appears as you type in the text field):**

Irrelevant 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Relevant

2. **Selecting appropriate suggestions from the autocompletion menu:**

Irrelevant 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Relevant

3. **Please briefly explain why you skipped the task?**

A.6 Post-task Class 2 Questionnaire: Summative Evaluation

1. **Finding ingredients using the toolbar (the bar in the left side of the screen):**

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

2. **Adding the ingredients found using the toolbar to the mashup:**

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

3. **Please briefly explain why you skipped the task?**

A.7 Post-task Class 3 Questionnaire: Summative Evaluation

1. **Being able to see the output mashup updated immediately after editing:**

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

2. **Being able to interact with the output mashup (like clicking and resizing and moving around widgets):**

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

3. **Getting highlighted text added to the text field as interacting with the output (i.e, clicking on a table results in adding and highlighting to the text “when an item is selected”):**

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

4. **Please briefly explain why you skipped the task?**

A.8 Post-task Class 4 Questionnaire: Summative Evaluation

1. Correcting your mistakes in the text field:

Difficult 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Easy

2. Text highlighting as you move your mouse on top of widgets and parts of text:

Useless 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ Helpful

3. Please briefly explain why you skipped the task?

Bibliography

- [1] S. Aghaee, M. Nowak, and C. Pautasso. Reusable decision space for mashup tool design. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 211–220, 2012.
- [2] S. Aghaee and C. Pautasso. Mashup development with html5. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, page 10, 2010.
- [3] S. Aghaee and C. Pautasso. The mashup component description language. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, pages 311–316, 2011.
- [4] S. Aghaee and C. Pautasso. Englishmash: Usability design for a natural mashup composition environment. In M. Grossniklaus and M. Wimmer, editors, *Current Trends in Web Engineering*, volume 7703, pages 109–120. Springer Berlin Heidelberg, 2012.
- [5] S. Aghaee and C. Pautasso. An evaluation of mashup tools based on support for heterogeneous mashup components. *Current Trends in Web Engineering*, pages 1–12, 2012.
- [6] S. Aghaee and C. Pautasso. Guidelines for efficient and effective end-user development of mashups. In *End-User Development*, pages 260–265. Springer Berlin Heidelberg, 2013.
- [7] S. Aghaee and C. Pautasso. Live mashup tools: Challenges and opportunities. In *Live Programming (LIVE), 2013 1st International Workshop on*, pages 1–4, 2013.
- [8] S. Aghaee and C. Pautasso. End-User Development of Mashups with NaturalMash. *Journal of Visual Languages & Computing*, 2014.

- [9] S. Aghaee, C. Pautasso, and A. De Angeli. Natural end-user development of web mashups. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 111–118, 2013.
- [10] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. Web services. In *Web Services*. Springer Berlin Heidelberg, 2004.
- [11] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: a data mashup fabric for intranet applications. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1370–1373. VLDB Endowment, 2007.
- [12] C. Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006.
- [13] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14:329–366, 2004.
- [14] U. Aßmann. *Invasive Software Composition*. Springer, 2003.
- [15] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.
- [16] M. A. Babar. *Software Architecture Knowledge Management: Theory and Practice*. Springer, 2009.
- [17] B. Beemer and D. Gregg. Mashups: a literature review and classification framework. *Future Internet*, 1(1):59–87, 2009.
- [18] M. Belaunde and S. B. Hassen. Service mashups using natural language and context awareness: A pragmatic architectural design. In *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, pages 404–411. IEEE, 2011.
- [19] D. Benslimane, S. Dustdar, and A. Sheth. Services mashups: The new generation of web applications. *Internet Computing, IEEE*, 12(5):13–15, 2008.
- [20] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations. *Software & Systems Modeling*, 11(3):431–461, 2012.

- [21] A. Bernstein, E. Kaufmann, C. Kiefer, and C. Bürki. SimPack: A Generic Java Library for Similiarity Measures in Ontologies. Technical report, Department of Informatics, University of Zurich, 2005.
- [22] P. A. Bernstein and L. M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, 2008.
- [23] A. Blackwell. Psychological Issues in End-User Programming. In H. Lieberman, F. Patern, and V. Wulf, editors, *End User Development*, volume 9 of *Human-Computer Interaction Series*, pages 9–30. Springer Netherlands, 2006.
- [24] M. M. Burnett. Visual programming. In J. G. Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*. Wiley, 1999.
- [25] C. Cappiello, M. Matera, M. Picozzi, G. Sprega, D. Barbagallo, and C. Francalanci. Dashmash: a mashup environment for end user development. In *Web Engineering*, pages 152–166. Springer, 2011.
- [26] S. Casteleyn, F. Daniel, P. Dolog, and M. Matera. *Engineering Web Applications*. Springer Publishing Company, Incorporated, 2009.
- [27] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, Mar. 1997.
- [28] O. Chudnovskyy, T. Nestler, M. Gaedke, F. Daniel, J. I. Fernández-Villamor, V. Chepegin, J. A. Fornas, S. Wilson, C. Kögler, and H. Chang. End-user-oriented telco mashups: the omelette approach. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 235–238. ACM, 2012.
- [29] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003.
- [30] A. Cypher and D. C. Halbert. *Watch what I do: programming by demonstration*. The MIT Press, 1993.
- [31] M. Csikszentmihalyi. Flow: The psychology of optimal experience. *Praha: Lidové Noviny*, 1990.
- [32] J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI '01 Extended Abstracts on Human Factors in Computing Systems*, pages 317–318, 2001.

- [33] F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan. Hosted universal composition: Models, languages and infrastructure in mashart. In *Conceptual Modeling-ER 2009*, pages 428–443. Springer, 2009.
- [34] F. Daniel, A. Koschmider, T. Nestler, M. Roy, and A. Namoun. Toward process mashups: Key ingredients and open research challenges. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, pages 9:1–9:8, 2010.
- [35] A. De Angeli, A. Battocchi, S. R. Chowdhury, C. Rodriguez, F. Daniel, and F. Casati. End-user requirements for wisdom-aware eud. In *Proceedings of the Third International Conference on End-user Development*, pages 245–250, 2011.
- [36] P. de Vrieze, L. Xu, A. Bouguettaya, J. Yang, and J. Chen. Process-oriented enterprise mashups. In *Grid and Pervasive Computing Conference, 2009. GPC'09. Workshops at the*, pages 64–71, 2009.
- [37] G. Di Lorenzo, H. Hacid, H.-y. Paik, and B. Benatallah. Data integration in mashups. *ACM Sigmod Record*, 38(1):59–66, 2009.
- [38] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi. Intel mash maker: join the web. *ACM SIGMOD Record*, 36(4):27–33, 2007.
- [39] R. Ennals and D. Gay. User-friendly Functional Programming for Web Mashups. *SIGPLAN Not.*, 42(9):223–234, Oct. 2007.
- [40] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.
- [41] M. Feldmann, T. Nestler, K. Muthmann, U. Jügel, G. Hübsch, and A. Schill. Overview of an End-user Enabled Model-driven Development Approach for Interactive Applications Based on Annotated Services. In *Proceedings of the 4th Workshop on Emerging Web Services Technology*, pages 19–28, 2009.
- [42] O. Ferschke, J. Daxenberger, and I. Gurevych. A survey of nlp methods and resources for analyzing the collaborative writing process in wikipedia. In I. Gurevych and J. Kim, editors, *The People Web Meets NLP, Theory and Applications of Natural Language Processing*, pages 121–160. Springer Berlin Heidelberg, 2013.

- [43] G. Fischer, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev. Meta-design: A Manifesto for End-user Development. *Commun. ACM*, 47(9):33–37, Sept. 2004.
- [44] T. Fischer, F. Bakalov, and A. Nauerz. An Overview of Current Approaches to Mashup Generation. In *Wissensmanagement*, pages 254–259, 2009.
- [45] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- [46] R. Fox, J. Cooley, and M. Hauswirth. Collaborative development of trusted mashups. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications Services*, pages 255–262, 2010.
- [47] E. M. Goncalves da Silva. *User-centric Service Composition - Towards Personalised Service Composition and Delivery*. PhD thesis, University of Twente, Enschede, May 2011.
- [48] L. Grammel and M.-A. Storey. An end user perspective on mashup makers. Technical Report DCS-324-IR, University of Victoria, September 2008.
- [49] C. Green et al. A summary of the psi program synthesis system. In *Proceedings of the 5th International Conference on Artificial Intelligence*, volume 1, pages 380–381, 1977.
- [50] D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009)*, in proceedings of WWW (International World Wide Web Conferences), 2009.
- [51] D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards Physical Mashups in the Web of Things. In *Proceedings of the 6th International Conference on Networked Sensing Systems*, pages 196–199, 2009.
- [52] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1–8, 2010.
- [53] A. Y. Halevy, N. Ashish, D. Bitton, M. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka. Enterprise Information Integration: Successes, Challenges and Controversies. In *Proc. of SIGMOD*, 2005.

- [54] J. J. Hanson. *Mashups: Strategies for the Modern Enterprise*. Addison-Wesley Professional, 2009.
- [55] B. Hartmann, L. Wu, K. Collins, and S. R. Klemmer. Programming by a sample: rapidly creating web applications with d. mix. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 241–250. ACM, 2007.
- [56] G. E. Heidorn. Automatic programming through natural language dialogue: A survey. *IBM Journal of Research and Development*, 20(4):302–313, 1976.
- [57] C. Hirsch, J. Hosking, and J. Grundy. Viki-builder: End-user specification and generation of visual wikis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 13–22, 2010.
- [58] D. D. Hoang, H.-y. Paik, and B. Benatallah. An analysis of spreadsheet-based services mashup. In *Proceedings of the Twenty-First Australasian Conference on Database Technologies-Volume 104*, pages 141–150. Australian Computer Society, Inc., 2010.
- [59] J.-M. Hoc and A. Nguyen-Xuan. Language semantics, mental models and analogy. *Psychology of programming*, 10:139–156, 1990.
- [60] V. Hoyer and M. Fischer. Market overview of enterprise mashup tools. In *Service-Oriented Computing-ICSOC 2008*, pages 708–721. Springer, 2008.
- [61] V. Hoyer, F. Gilles, T. Janner, and K. Stanoevska-Slabeva. SAP Research Rooftop Marketplace: Putting a Face on Service-Oriented Architectures. In *Proceedings of the 2009 Congress on Services - I*, pages 107–114, 2009.
- [62] V. Hoyer, K. Stanoevska-Slabeva, S. Kramer, and A. Giessmann. What are the business benefits of enterprise mashups? In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–10, 2011.
- [63] D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. *Web Semant.*, 5(1):16–27, Mar. 2007.
- [64] M. Imran, F. Kling, S. Soi, F. Daniel, F. Casati, and M. Marchese. Reseval mash: a mashup tool for advanced research evaluation. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 361–364. ACM, 2012.

- [65] M. Imran, S. Soi, F. Kling, F. Daniel, F. Casati, and M. Marchese. On the systematic development of domain-specific mashup tools for end users. In *Proceedings of the 12th International Conference on Web Engineering*, pages 291–298, 2012.
- [66] R. Jeffries and J. Rosenberg. Comparing a form-based and a language-based user interface for instructing a mail program. *ACM SIGCHI Bulletin*, 17(SI):261–266, 1986.
- [67] M. C. Jones, E. F. Churchill, and M. B. Twidale. Mashing up visual languages and web mash-ups. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 143–146, 2008.
- [68] E. Kaufmann and A. Bernstein. How useful are natural language interfaces to the semantic web for casual end-users? In *The Semantic Web*, pages 281–294. Springer, 2007.
- [69] S. Kent. Model driven engineering. In *Integrated Formal Methods*, pages 286–298, 2002.
- [70] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21, 2011.
- [71] R. R. Korfhage and M. A. Korfhage. Criteria for iconic languages. In *Visual languages*, pages 207–231. Plenum Press, 1986.
- [72] A. Koschmider, V. Torres, and V. Pelechano. Elucidating the mashup hype: Definition, challenges, methodical guide and tools for mashups. In *Proceedings of the 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web at WWW*, 2009.
- [73] C. Letondal. Participatory programming: Developing programmable bioinformatics tools for end-users. In H. Lieberman, F. Patern, and V. Wulf, editors, *End User Development*, volume 9 of *Human-Computer Interaction Series*, pages 207–242. Springer Netherlands, 2006.
- [74] H. Lieberman, F. Paternò, M. Klann, and V. Wulf. End-user development: An emerging paradigm. In *End user development*, pages 1–8. Springer, 2006.

- [75] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 97–106. ACM, 2009.
- [76] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 943–946, 2007.
- [77] J. López, F. Bellas, A. Pan, and P. Montoto. A Component-Based Approach for Engineering Enterprise Mashups. In *Proceedings of the 9th International Conference on Web Engineering*, pages 30–44, 2009.
- [78] P. Lubbers and B. Albers. Harnessing the power of HTML5 web sockets to create scalable real-time applications presentation. *Web2.0 Expo SF*, 2010.
- [79] A. MacLean, K. Carter, L. Löfstrand, and T. Moran. User-tailorable systems: Pressing the issues with buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 175–182, 1990.
- [80] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 15–23, 2010.
- [81] Z. Maraïkar, A. Lazovik, and F. Arbab. Building mashups for the enterprise with sabre. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 70–83, 2008.
- [82] E. M. Maximilien, A. Ranabahu, and K. Gomadam. An online platform for web apis and service mashups. *IEEE Internet Computing*, 12(5):32–43, Sept. 2008.
- [83] E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A domain-specific language for web apis and services mashups. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC 2007)*, pages 13–26. Springer, 2007.
- [84] A. Mørch. Computers and design in context. chapter Three Levels of End-user Tailoring: Customization, Integration, and Extension, pages 51–76. MIT Press, 1997.

- [85] A. I. Mørch. Designing for radical tailorability: coupling artifact and rationale. *Knowledge-Based Systems*, 7(4):253–264, 1994.
- [86] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.
- [87] B. A. Myers, A. J. Ko, and M. M. Burnett. Invited research overview: End-user programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, pages 75–80, 2006.
- [88] A. Namoun, T. Nestler, and A. De Angeli. Service composition for non-programmers: Prospects, problems, and design recommendations. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS)*, pages 123–130, 2010.
- [89] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. The MIT Press, 1993.
- [90] T. Nestler, M. Feldmann, G. Hübsch, A. Preußner, and U. Jugel. The servface builder—a wysiwyg approach for building service-based applications. In *Web Engineering*, pages 498–501. Springer, 2010.
- [91] D. A. Norman and S. W. Draper. *User centered system design; new perspectives on human-computer interaction*. L. Erlbaum Associates Inc., 1986.
- [92] M. Nowak and C. Pautasso. Goals, questions and metrics for architectural decision models. In *Proceedings of the 6th International Workshop on SHaring and Reusing Architectural Knowledge*, pages 21–28, 2011.
- [93] T. O'Reilly. What is Web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, (1):17, 2007.
- [94] C. Pautasso and G. Alonso. The jopera visual composition language. *Journal of Visual Languages & Computing*, 16(1):119–152, 2005.
- [95] S. Pietschmann, V. Tietz, J. Reimann, C. Liebing, M. Pohle, and K. Mei. A Metamodel for Context-aware Component-based Mashup Applications. In *Proceedings of the 12th International Conference on Information Integration & Web-based Applications & Services*, pages 413–420, 2010.
- [96] S. Pietschmann, M. Voigt, A. Rumpel, and K. Meißner. Cruise: Composition of rich user interface services. In *Web Engineering*, pages 473–476. Springer, 2009.

- [97] H. Prähofer, D. Hurnaus, and H. Mössenböck. Building end-user programming systems based on a domain-specific language. In *6th OOPSLA Workshop on Domain-Specific Modeling*, page 33, 2006.
- [98] A. Repenning and A. Ioannidou. What makes end-user development tick? 13 design guidelines. In *End User Development*, pages 51–85. Springer, 2006.
- [99] L. Richardson and S. Ruby. *RESTful web services*. O'Reilly, 2008.
- [100] M. Sabbouh, J. Higginson, S. Semy, and D. Gagne. Web mashup scripting language. In *Proceedings of the 16th international conference on World Wide Web*, pages 1305–1306. ACM, 2007.
- [101] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.
- [102] M. Schrenk. *Webbots, spiders, and screen scrapers: A guide to developing Internet agents with PHP/CURL*. No Starch Press, 2012.
- [103] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.
- [104] I. Smith. Doing web clippings in under ten minutes. Technical report, Intranet Journal, March 2001.
- [105] V. Stirbu. Towards a restful plug and play experience in the web of things. In *Semantic Computing, 2008 IEEE International Conference on*, pages 512–517, 2008.
- [106] A. Strunk. Qos-aware service composition: A survey. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS)*, pages 67–74. IEEE, 2010.
- [107] S. L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.
- [108] J. C. Thomas and J. D. Gould. A psychological study of query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 439–445. ACM, 1975.

- [109] R. Tuchinda, C. A. Knoblock, and P. Szekely. Building mashups by demonstration. *ACM Transactions on the Web (TWEB)*, 5(3):16, 2011.
- [110] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
- [111] K. Vredenburg, J.-Y. Mao, P. W. Smith, and T. Carey. A survey of user-centered design practice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 471–478, 2002.
- [112] G. Wang, S. Yang, and Y. Han. Mashroom: end-user mashup programming using nested tables. In *Proceedings of the 18th international conference on World wide web*, pages 861–870. ACM, 2009.
- [113] E. Wenger. *Communities of practice: Learning, meaning, and identity*. Cambridge university press, 1998.
- [114] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444. ACM, 2007.
- [115] S. D. Wright et al. Designing mashups with excel and visio. In *Expert SharePoint 2010 Practices*, pages 513–539. Apress, 2011.
- [116] J. Yang, J. Han, X. Wang, and H. Sun. Mashstudio: An on-the-fly environment for rapid mashup development. In *Proceedings of the 5th International Conference on Internet and Distributed Computing Systems*, pages 160–173, 2012.
- [117] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *IEEE Internet Computing*, 12(5):44–52, Sept. 2008.
- [118] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In *Proceedings of the 16th international conference on World Wide Web*, pages 923–932. ACM, 2007.
- [119] J. Zhou, M. Wang, and H. Zhao. Enterprise information integration: State of the art and technical challenges. In *Knowledge Enterprise: Intelligent Strategies in Product Design, Manufacturing, and Management*. Springer US, 2006.

