Reusable Decision Space for Mashup Tool Design

Saeed Aghaee
Faculty of Informatics,
University of Lugano (USI)
via Buffi 13, 6900 Lugano,
Switzerland
saeed.aghaee@usi.ch

Marcin Nowak
Faculty of Informatics,
University of Lugano (USI)
via Buffi 13, 6900 Lugano,
Switzerland
marcin.nowak@usi.ch

Cesare Pautasso
Faculty of Informatics,
University of Lugano (USI)
via Buffi 13, 6900 Lugano,
Switzerland
c.pautasso@ieee.org

ABSTRACT

Mashup tools are a class of integrated development environments that enable rapid, on-the-fly development of mashups—a type of lightweight Web applications mixing content and services provided through the Web. In the past few years there have been growing number of projects, both from academia and industry, aimed at the development of innovative mashup tools. From the software architecture perspective, the massive effort behind the development of these tools creates a large pool of reusable architectural decisions from which the design of future mashup tools can derive considerable benefits. In this paper, focusing on the design of mashup tools, we explore a design space of decisions comprised of design issues and alternatives. The design space knowledge not only is broad enough to explain the variability of existing tools, but also provides a road-map towards the design of next generation mashup tools.

Author Keywords
Mashup tools; software architecture; design rationale;

ACM Classification Keywords
D.2.2 Software Engineering: Design

INTRODUCTION

Mashup tools are interactive systems which target the needs of end-users developing a specific kind of Web applications, built of the reuse and composition of multiple Web data sources and Web services [5]. The past few years have witnessed a rapid growth of many interactive Mashup tools, both from research and industry, offering a broad range of characteristics and affordances. Some are based on visual composition languages [30], others feature a high degree of automation and liveness [47], many support collaborative development [20], engaging public and private online communities.

The challenges faced by mashup tools designers include the need for defining a high level descriptions of computations and integration logic to be combined with suitable abstractions to represent Web widgets, distributed Web services and Web data sources as reusable components [14]. Mashups can be and are built by programmers using traditional Web technologies and tools [1], as shown on ProgrammableWeb[1], a directory listing thousands of mashups and Web APIs. The goal of most mashup tools is to enable non-programmers to build mashups, by making it easy [8] to quickly [29] reuse and reassemble whatever content, services, APIs and data sources can be found on the Web.

Architectural knowledge management [3] advocates the extraction of design knowledge from successful software projects in order to accumulate best practices [6], design patterns [7] and reusable architectural decisions. In this paper we survey the existing mashup tooling landscape with the goal of harvesting reusable architectural design decisions. The goal is to take a conscious approach to explicit design decisions, which is intended to result in higher quality software architectures. The decisions are structured into a design space, which (1) helps to classify and explain the heterogeneity of existing mashup tools; and (2) by enumerating relevant design issues and their dependencies, provides a valuable guidance model to mashup tool designers.

The rest of this paper is structured as follows: in the next section we describe the methodology we applied to extract the design issues and alternatives as well as to conduct our survey of mashup tools based on them. Next, we present the core of this paper consisting of the issues and alternatives that arise in the design of a selection of 22 mashup tools. Afterwards, we discuss how the issues and alternatives relate to each other in the context of the design of existing tools. Finally, the presentation of the related work is followed by the conclusions.

METHODOLOGY

In order to give a clear structure to the content collected in this paper, we have constructed a model (Figure 1) conforming to a simple, yet powerful decision metamodel proposed in [43] and used the corresponding tool[2] to gather and process the knowledge. The metamodel 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, there are more than 60 mashup tools, from which we picked 22 tools based on their availability and their

EICS'12, June 25–26, 2012, Copenhagen, Denmark.
Copyright 2012 ACM 978-1-4503-1168-7/12/06...$10.00.

[1] `http://www.programmableweb.com/`
[2] `http://saw.inf.unisi.ch`

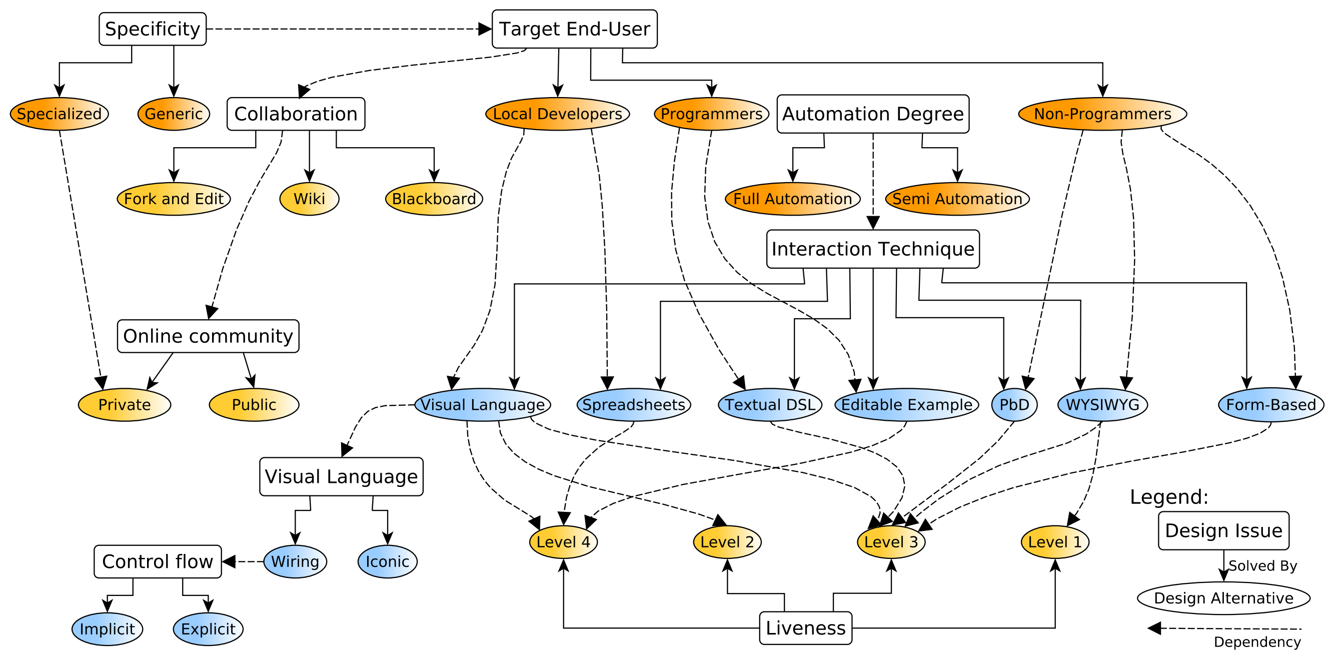

Figure 1. Mashup tool design dpace overview

representativeness to ensure that at least a concrete example corresponds to each design alternative. We further refined the knowledge into design decisions by eliminating overlapping issues and alternatives and dividing them into groups. The validation was conducted by checking whether the decision space covered all the issues found in the selected mashup tools. We prioritized those design issues with the most impact on the usability of mashup tools, seen as a specific class of interactive systems.

For the sake of readability, we classified the 9 design issues and 27 alternatives in three different groups (as shown colored in Figure 1): (1) Strategic (red), grouping the top level criteria regarding the user community where a mashup tool is supposed to be deployed; (2) Environment-specific (orange), collecting issues on the development environment where users are provided with facilities to develop and possibly execute and debug mashups; (3) Language (blue), concerning the design of mashup composition languages.

STRATEGIC DESIGN ISSUES FOR MASHUP TOOLS

Design Issue: Specificity

Mashups have gained a broad range of applications ranging from daily utilities of Web users to narrowly specialized domains. Soi and Baez [46], therefore, distinguish between specialized and generic mashup tools.

Alternative: Generic

The main characteristic of generic mashup tools is that they do not target any specific group of end-users or any particular application domain, but rather focus on addressing the daily needs of end-users. Generic mashup tools take advantage of publicly available Web resources. The growing number and big diversity of these resources enable the creation of innovative mashups that can be published, discovered and used by other end-users.

Benefits: A generic tool has the potential to reach a wide range of end-users and become popular on the Web. Depending on the architecture of the tool, it can be further tailored to fit a particular domain or application [46].
Challenges: Due to the diversity and sheer number of end-users on the Web, it is a non-trivial task to assess and support the needs and abilities (i.e., programming skills) of all.
Example: *FeedRinse*[3] exemplifies a generic mashup tool. With the use of this tool, users can filter and combine multiple RSS feeds, and republish the results in a single RSS feed.

Alternative: Specialized

Recently, investigating the applications of mashups in different specialized domains has gained momentum. Examples of the use of mashups in specialized domains include enterprise computing [29], e-learning [16], bioinformatics [21], and telecom [4]. Specialized mashup tools support the development of mashups suited for these domains by mixing domain-related resources that can be either free or proprietary. One example of such domain-related resources could be a private enterprise database.

Benefits: The end-users targeted by specialized tools are often limited in number, homogeneous in terms of technical skills, and geared towards the same task. These factors help assess, evaluate, and improve the usability of the tool.
Challenges: Designing a mashup tool for a specialized domain requires an advanced, in-depth knowledge about that

[3]`http://feedrinse.com/`

domain. The specific functional and non-functional requirements for mashups within the domain (e.g., secure data access for enterprise data fusion) need to be extracted so that a suitable tool for domain-specific mashups can be designed.
Example: *Kapow Katalyst*[4], and the tool presented by Capuano et. al. [10] exemplify two specialized mashup tools targeting, respectively, telecom and e-learning domains.

Design Issue: Target End-user
As described by Nardi [41, p. 104], in terms of technical and programming skills, there is a spectrum of end-users, ranging from non-programmers to experienced programmers. In the middle of this spectrum lie professional end-users without programming skills but interests in computer and programming, who are referred to by Nardi as "local developers". Determining which group of end-users is targeted by a mashup tool is a design issue, as the tool should provide different affordances for a different group of end-users.

Alternative: Non-Programmers
Non-programmer do not have programming skills. Yet, they are interested in creating mashup as long as that does not require them to learn and use a programming language.

Benefits: A mashup tool created for non-programmers is designed to suit those with minimum technical skills. As a result, it will be usable and applicable to not only non-programmers, which constitute the majority of Web users, but also both local developers and programmers.
Challenges: Non-programmers should be provided with tools that limit their involvement in the development process to small customizations of predefined mashup templates, or execution of parametrized mashups.
Example: *Dapper*[5] is designed for non-programmers. It provides a set of easy-to-use toolchains to visually scrape content from a website and integrate it with Web feeds, without the need to get involved in the development process.

Alternative: Local Developers
Local developers are those non-programmers who usually have advanced knowledge in computer tools [41, p. 104]. They are willing to explore and harvest all the functionality of a mashup tool tailored for their abilities.

Benefits: Tools targeting local developers can provide composition functionality, where mashups can be assembled from scratch by composing predefined components or by customizing and changing existing examples and templates.
Challenges: Mashup tools for local developers must provide a very high level of abstraction that ideally hides all the underlying technical complexity of the mashup development. However, such a high level of abstraction usually comes at the price of sacrificing the expressive power of the tool. The challenge is thus to find a proper balance between them.
Example: *JOpera* [45] offers a high expressive power to create various types of mashups. Despite the visual language and the level of abstraction provided by the tool, creation of some complex mashups may still involve a small amount of coding in JavaScript and HTML.

[4]`http://kapowsoftware.com/`
[5]`http://open.dapper.net/`

Alternative: Programmers
Programmers have adequate programming skills and experience to develop mashups using programming and scripting languages (e.g., JavaScript and PHP).

Benefits: Programmers can produce high quality, feature-rich, and useful mashups, which can be later reused by non-programmers. Tools targeting programmers can provide both composition and component development features. Components can be then provided to local developers and non-programmers for reassembly and customization.
Challenges: The expressive power of the tool should not be compromised when compared to existing general-purpose programming and Web scripting languages for developing mashups.
Example: *Swashup* [36] is a Web-based development environment for a textual Domain-Specific Language (DSL) based on the Ruby on Rails framework (RoR).

Design Issue: Automation Degree
A mashup tool needs to leverage automation to lower the barriers of and enable on-the-fly mashup development. The automation degree of a mashup tool, hence, refers to how much of the development process can be undertaken by the tool on behalf of its users, and can be broken down into *semi-automation* and *full automation*.

Alternative: Full Automation
Full automation of mashup development eliminates the need for direct involvement of users in the development process. Instead, the users will gain a supervisory role with the opportunity to provide input (i.e., requirements), intervene in the development process, and validate the final result.

Benefits: Since the development process is carried out by the tool, the burden of learning is considerably lifted from the users. Also, if designed properly, it can significantly decrease the effort of mashup development.
Challenges: A common challenge is not to deviate from the user's needs by producing irrelevant mashups. The tool can allow 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, users might encounter the risk of experiencing many iterations.
Example: *Piggy Bank* [27] 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. Still, their users are involved in the development process.

Benefits: Due to the direct involvement of the user in the development process, there is a lower probability of deviation from his/her needs, compared to automatic tools.
Challenges: The users should go through a relatively longer learning curve to be able to create their desired mashups. This presents a challenge to motivate and encourage end-users to learn how to use the tool.

Example: The majority of mashup tools are semi-automatic. RoofTop [25] automates and abstracts many complex aspects of mashup development. Yet, users need to actively select and connect widgets to create a mashup with it.

ENVIRONMENT-SPECIFIC DESIGN ISSUES

Design Issue: Liveness

In the context of visual languages, Tanimoto proposed the concept of *liveness* [47], 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 run-time system. A prototype mashup usually only has the final user interface without underlying functionality.

Benefits: The main benefit is the relative simplicity of these tools. They are only used to create prototype mashups by allowing to design their user interfaces in a visual manner.
Challenges: The goal of the majority of mashup tools is to provide a development environment for creating executable mashups. However, tools supporting liveness level 1 only help to create non-executable prototype mashups.
Example: Microsoft Visio enables the creation of prototype mashups. The resulting prototypes can be fed with data and executed by Microsoft Excel [52].

Alternative: Level 2. Executable flowchart

The second level of liveness is characterized by the fact that the mashup design blueprint carries sufficient details to give it an executable semantics.

Benefits: A primary benefit of having (indirectly) executable blueprints is that its consistency (logical, semantical, or syntactical) can be verified. Another advantage is that such blueprint is self-contained in terms of documentation, hence can serve as reference for users and developers.
Challenges: The fact that design 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.
Example: *Petals BPM*[6] is a Business Process Modeling Notation (BPMN) modeler that offers features such as validation, and allows the created diagrams to be exported to WS-BPEL format for the sake of execution using a different tool.

Alternative: Level 3. Edit triggered updates

Mashups characterized by 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.

Benefits: Thanks to this feature, mashup designers and developers are released from the burden of going through a potentially time-consuming manual deployment process.
Challenges: Users need to be aware in which mode (design-time editor or run-time execution monitor) they are operating the mashup tool. 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.
Example: A good example of a mashup tool of liveness level three is *JackBe Presto*[7]. 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.

Benefits: Designers 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 cycle is very rapid.
Challenges: 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, a challenge which designers need to face is that highly responsive environments can result in high costs of running the mashup, as – for example – remote Web services need to be invoked every time a change is done on the mashup code.
Example: *DashMash* [9] supports liveness at level 4 by merging the mashup design and runtime environments, and proving a mechanism to keep both of them synchronized.

Design Issue: Online Community

Online communities are an important resource in assisting end-users to program [41]. They can potentially support technical discussion as well as collaborative mashup categorization, sharing, rating, and recommendation [22]. 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 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.

Benefits: The added value of a public community lies in its great potential for growth, ultimately resulting in the increased number of the tool users.
Challenges: As the content shared in the community is public, users may refuse or refrain from sharing certain details.
Example: *Yahoo! Pipes*[8] 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.

[6]`http://research.petalslink.org/display/petalsbpm/`

[7]`http://www.jackbe.com/enterprise-mashup/`

[8]`http://pipes.yahoo.com/`

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 users.

Benefits: The content stored in private communities is inaccessible to non-members, resulting in a higher level of confidence for users to discuss issues related to their organization.
Challenges: Private communities require much more effort to start. Content should be mostly created by the community staff, since with a small number of users, there will not be much user-generated content initially.
Example: *IBM Mashup Center*[9] allows enterprises to build their own private community, organized around a centralized catalogue. Users can publish mashups to this catalogue so that other users can discover and reuse them.

Design Issue: Collaboration

From a software development methodology point of view, mashup development is a form of agile development [26], which is characterized by a high degree of collaboration between the involved actors. Mashup development can also be performed in a collaborative manner [15], provided that it is supported by the mashup tool. To this end the availability of on online community is both essential and beneficial. To date, this support has came in different forms which we refer to as *fork and edit*, *wiki*, and *blackboard*.

Alternative: Fork and Edit

Fork and edit is a common method for enabling collaborative mashup development that is typically based on online communities. The method relies on a scenario, where a user creates and shares a mashup within the tool community that is later found by another user, who then edits a copy of it or reuses it inside a new mashup, and finally shares the resulting mashup back to the same community so that it can be further modified and recursively embedded into other mashups.

Benefits: This method encourages reuse and sharing amongst users. It also eliminates any chance of version or instance conflicts, due to the fact that each different instance derived from the same mashup is associated with a different user.
Challenges: Mashup instance duplication, i.e., storing multiple copies of the same mashup associated to different users, should be prevented. Another challenge is that fork and edit-based system inherently do not provide a straightforward way to merge two or more mashups originated from the same mashup into a single one.
Example: *Yahoo! Pipes* is an example of a collaborative tool following the fork and edit/reuse method. The tools is equipped with a large online community that offers mashup sharing, search, and cloning features.

Alternative: Wiki

Collaborative mashup development can be enabled with the use and adaptation of the wiki method, whose main features include versioning, multi-author editing, and changelogs.

[9] `http://www.ibm.com/software/info/mashup-center/`

Benefits: The working copy of a mashup is always writeable because all changes are local until committed. Moreover, commits are atomic, i.e., either all or no changes are committed. Another clear benefit is the version history allowing a user to track changes and revert back to earlier versions.
Challenges: Storing and keeping track of different versions of a mashup created by a tool may require lots of space. The challenge concerning users is that they may find it difficult to avoid and resolve editing conflicts.
Example: *Lively Wiki* [33] is a collaborative mashup tool based on the wiki method. It combines the wiki principles with a direct-manipulation user interface, through which users can create and edit mashups.

Alternative: Blackboard

A blackboard-based environment manages collaborative development in a realtime basis. All the involved users can observe changes to the mashup at the time they happen.

Benefits: The overall collaborative development process is much faster, since changes made by users take effect immediately without requiring any intermediate action (e.g., commit, or publish). As a result, conflicts are not encountered.
Challenges: The consistency and validity of the mashup need to be ensured. It can be addressed by providing versioning and history tracking. Another technical challenge is also to minimize the communication latency amongst the participants so as to ensure a real-time experience.
Example: *Sqwelch* [20] is a semantically-enabled mashup tool that allows blackboard-like collaboration amongst its users to create mashups. This mashup tool does not support versioning and history tracking.

LANGUAGE-LEVEL DESIGN ISSUES

Design Issue: Interaction Technique

There have been a number of interaction techniques through the use of which the barriers of programming can be lifted from end-users [39]. We list below some representative techniques which have been used by mashup tools. Some tools are known for using multiple techniques in combination.

Alternative: Textual DSL

Domain Specific Languages (DSL [19]) are languages targeted to address specific problems in a particular domain. Textual DSLs define textual syntax, that may or may not resemble an existing general-purpose programming language.

Benefits: DSLs, particularly those built internally on top of a general purpose programming language, usually offer a high expressive power.
Challenges: In terms of learning barriers, textual DSLs are similar to programming languages.
Example: *Swashup* [36] is a textual DSL for the mashup domain built on top of the Ruby on Rails framework.

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 [38].

Benefits: 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 [37], e.g. showing both the design and runtime environments in the same screen.
Challenges: 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.
Example: SABRE [35] is based on a visual language corresponding to Reo [2], a coordination language that is used to define the logic of the mashup.

Alternative: What-You-See-Is-What-You-Get
In the context of mashups, WYSIWYG (What You See Is What You Get) enables 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.

Benefits: Since users always see the resulting mashup, the whole development process might be streamlined. Another potential benefit is the increase of the tool directness. Users place visual objects exactly in the places where they are meant to be during the runtime.
Challenges: 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 WYSIWYG tool.
Example: ServFace Builder [42] 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: Programming by Demonstration
As opposed to direct programming, PbD (Programming by Demonstration) suggests to teach a computer by example how to accomplish a particular task [11].

Benefits: This is a powerful technique that helps remove much of programming barriers. Users demonstrate what is the mashup they want without worrying about how it should be programmatically implemented.
Challenges: 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 [41].
Example: Karma [49] allows users to create data mashups interactively by providing examples demonstrating the integration of data from different websites.

Alternative: Programming by Example Modification
Another powerful technique to remove the burden of programming is to let users modify and change the behavior of existing examples, instead of programming from scratch [34].

Benefits: 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.
Challenges: 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.
Example: *d.mix* [23] allows users 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.

Alternative: Spreadsheets
Spreadsheets are one of the most popular and widely used end-user programming approaches to store, manipulate, and display complex data.

Benefits: Since the majority of mashups are about data integration, manipulation and visualization, spreadsheets can potentially be used as a natural approach to this end.
Challenges: Spreadsheets can not be used to design the user interface of a mashup.
Example: *Husky*[10] is a spreadsheet-based service composition and mashup development tool. Each cell in the spreadsheet represents a service or data source.

Alternative: Form-based
In form-based interaction, users are asked to fill out a form to create a new or change the behavior of an existing object.

Benefits: 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 [48].
Challenges: Form-based tools cannot handle complex composition patterns for mashups [28].
Example: *FeedRinse* provides a form-based mechanism to filter and aggregate Web feeds.

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.

Benefits: 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.
Challenges: 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.

[10]`http://www.husky.fer.hr/`

Example: The visual language incorporated by *MashArt* [12] 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.

Benefits: Properly designed icons are generally easily interpreted, understood and remembered by users.
Challenges: An iconic visual language requires to invest significant effort and thought into icon design [31]. This is essential to avoid any further changes to the appearance of the icons, which causes confusion due to unexpected behavior.
Example: *VikiBuilder* [24] 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.

Design Issue: Control Flow

In a visual programming language, just like in any other programming language, there should be a method with which the user can define the program's control flow. In case of the wiring method, this can be achieved through the use of either *explicit* methods or *implicit* methods.

Alternative: Explicit

The control flow is explicitly defined, for instance, by adding directed arrows connecting the boxes, or putting the boxes in a specific order (e.g., from left to right) that corresponds to their execution order.

Benefits: This alternative gives much more control over the development of the application logic of a mashup. Process-oriented mashups, for example, often incorporate a relatively complex application logic [50].
Challenges: Data flow and control flow graphs should be defined separately, which poses extra barriers on the development process. Additionally, representing a mashup with more than one diagram, each corresponding to either data flow graph or control flow graphs of the mashup, can impair the understandability and readability of the language.
Example: *JOpera* supports explicit design of the control flow graph. Each box in a control flow graph, represents an executable task whose incoming and outgoing edges define the control flow to and from the task.

Alternative: Implicit

In this case, the control flow of a mashup is derived from its data flow graph. For instance, in the simplest case, the flow of data/message from one activity to another suggests the same flow of control between them.

Benefits: This method is lightweight and gives users a natural way to represent parallel execution. They are only required to design the flow of data/message between components, which also declares a partial execution order between them.
Challenges: The shortcoming of this method is that it can only be used to create mashups with simple control flow patterns, such as those that do not contain branches and loops.

Example: *Proto Financial*[11] is based on a visual wiring language that allows integration of heterogeneous data sources in an enterprise setting.

DISCUSSION

Table 1 summarizes the mashup tool design space as well as the decisions made by the designers of each tool pertaining each design issues we previously defined within this decision space, which is mostly focused on the design of interactive systems. Given the table, we attempt to (1) analyze the design issues with respect to the evolution of the decisions made by different tools over the last 6 years; (2) highlight and scrutinize the most and the least frequently chosen design alternatives; and (3) present the impacts of design decisions on other issues and alternatives within the space.

Regarding the issue of specificity, the first-generation mashup tools are commonly general purpose, for instance, Yahoo! Pipes and Dapper. Interestingly, the application of mashups in various domains has resulted in an increase in the number of specialized tools, which have started to appear more recently. This, however, does not imply a trend shifting from generic tools to specialized tools. Rather, the broad and emerging application domains for mashups provide an on-going demand for the design of tailored mashup tools targeting different and specific domains of application.

Both generic and specialized mashup tools, in the majority of cases, tend to target non-programmers. This is a safe strategy especially for generic mashup tools, whose target users' population is large, unknown, and is dominated by non-programmers. Conversely, the target users of a specialized tool can be determined through a technical skills assessment process, for example.

After the end-user group targeted by a tool are assessed on technical skills, an important decision to make concerns the degree of the automation of the tool. Fully automatic tools are commonly believed to best serve non-programmers [18]. However, the problem lies in how a user is supposed to communicate the requirements for the mashup to be built with an automatic tool and also to give guidance and feedback to the tool in order to iteratively converge on the desired outcome. This may be done using an interaction technique as complicated as, or even more complicated than the one used to program a mashup with a semi-automatic tool. The risk of a fully automatic tool becoming too difficult to use has possibly led the design of most tools in our survey to abstain from choosing the full automation design alternative.

The core design issue for a mashup tool is to select suitable interaction techniques to let mashup developers communicate the mashup composition logic and, in case of a fully automatic tool, the goal and requirements of the mashup. To do so, there is a pool of available interaction techniques, which can also be combined in case of hybrid tool designs. According to the results of our survey, the majority of semi-automatic tools facilitate multiple interaction techniques. The most popular combination of interaction techniques is visual language and form-based programming, where tools typically offer a

[11]`http://www.protosw.com/`

Name	Piggy Bank [27] (2005)	FeedRinse (2006)	Dapper (2006)	JackBe Presto (2006)	Swashup [36] (2007)	d.mix [23] (2007)	JOpera [45] (2007)	Yahoo! Pipes (2007)	Karma (2008)	SABRE [35] (2008)	Kapow Katalyst (2009)	MashArt [12] (2009)	Lively Wiki [33] (2009)	RoofTop [25] (2009)	IBM Mashup Center (2009)	ServFace Builder [42] (2010)	VikiBuilder [24] (2010)	Microsoft Visio [52] (2010)	Husky (2011)	DashMash [9] (2011)	Petals BPM (2011)	Sqwelch [20] (2011)
Specificity																						
Generic	+	+	+		+	+	+	+	+									+	+	+		
Specialized				+						+	+	+	+	+	+	+	+				+	+
Target End-User																						
Local Developers				+			+				+											
Non-Programmers	+	+	+			+		+	+	+		+	+	+	+	+	+	+	+	+	+	+
Programmers					+																	
Automation Degree																						
Full Automation	+																					
Semi Automation		+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Liveness																						
Level 1																		+				
Level 2																					+	
Level 3	+	+	+	+	+	+	+	+	+	+	+	+			+	+	+		+			+
Level 4													+	+						+		
Online Community																						
Private				+										+	+							
Public		+						+			+											
Collaboration																						
Blackboard																						+
Fork and Edit				+				+							+							
Wiki													+									
Interaction Technique																						
Editable Example						+																
Form-based		+		+			+	+			+	+			+	+	+					+
PbD			+						+													
Spreadsheets									+										+			
Textual DSL				+	+																	
Visual Language				+			+	+		+	+	+		+	+		+			+	+	
WYSIWYG													+	+	+	+		+				+
Visual Language																						
Iconic										+							+					
Wiring				+			+	+			+	+		+	+					+	+	
Control Flow																						
Explicit							+				+										+	
Implicit				+				+				+		+	+					+		

Table 1. Summary of the mashup tool design decisions over the mashup design space

visual wiring language, in which graphical boxes representing data sources, processing operators, etc. either contain or associated with a configuration form to specify and control the configuration properties associated with the box.

In spite of the popularity of visual languages based on the wiring paradigm, according to a study conducted by Namoun et. al. [40], 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. Thereby, forcing users to explicitly define flow of control in addition to flow of data in a mashup (i.e., explicit control flow alternative) can end up adding even more cognitive costs to learn how to interact with the mashup tool.

One of the most promising approaches to lower cognitive costs and increase motivation is to facilitate collaborative development [17]. In doing so, establishing an online community is crucial to effective collaboration as it brings together users with similar interests and common ground [51]. It should also be noted that the full potential of an online community to this end is exploited only when it is internally built upon the actual 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 FeedRinse and Kapow Katalyst). Therefore, it is important to consider collaborative development and internal communities designed for the sake of collaboration as important features to be implemented in next generation mashup tools.

Liveness is also another important issue affecting the usability of a tool. Interestingly, the vast majority of the tools already support liveness at level 3 through a "run" button that takes the users to the runtime environment where the mashup will be deployed and executed. The assumption is that users are capable to distinguish between the design-time modeling and composition environment and its run-time version, where the mashup execution occurs. A few tools (e.g., DashMash and RoofTop) have begun to remove this artificial distinction, and in the future we expect to witness the proliferation of mashup tools supporting the highest level of liveness, possibly combined with a WYSIWYG interface.

RELATED WORK

The field of mashup development has matured to a level where some frequently used patterns for mashup design have emerged (e.g., [44]). This paper makes a contribution towards a design space for mashup *tools* design, focusing on a collection of design issues which impact on usability aspects of mashup tools, seen as a class of interactive systems.

Existing surveys on mashup tools have been published focusing on different aspects, which can contribute together with this paper to build an even larger design space. For example, [18] classifies a small number of tools according to the chosen programming technique. The result is that no surveyed tool completely satisfies the needs for end-users. [13] collect examples of an a specific kind of mashup tools, performing integration at the business process level. A number of domain-specific, enterprise mashup tools have been classified in 2008. The survey published in [32] aims at classifying mashup tools according to a set of run-time deployment issues (e.g., client-side vs. server-side deployment, or user interface vs. data vs. process-level integration) which are complementary to the ones collected in this paper.

CONCLUSION

In this paper, we have collected 9 design issues and 27 reusable design alternatives covering essential aspects of the design space for mashup tools. To build this space, we have reconstructed and analyzed the design decisions taken by the creators of 22 contemporary mashup tools. The accumulated architectural knowledge is a useful reference and survey for engineering interactive systems for mashup composition. First, tool designers can use this survey to foster innovation during the design of next generation solutions. Second, the comprehensive explanation of the heterogeneity of mashup tools presented in this paper can provide researchers with a novel design-centric view over the state of the art. All in all, the goal of this survey is to promote the reuse of good design solutions, thus improving both the quality and the efficiency of the design process for next generation mashup tools.

ACKNOWLEDGEMENT

The work presented in this paper has been supported by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22 127386) and CLAVOS project (Grant Nr. 125337).

REFERENCES

1. Aghaee, S., and Pautasso, C. Mashup development with html5. In *Proc. of Mashups '09/'10* (2010).
2. Arbab, F. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci. 14* (2004), 329–366.
3. Babar, M. A., Dingsøyr, T., Lago, P., and van Vliet, H. *Software Architecture Knowledge Management - Theory and Practice*. Springer, 2009.
4. Banerjee, N., and Dasgupta, K. Telecom mashups: enabling web 2.0 for telecom services. In *Proc. of ICUIMC 2008* (2008).
5. Benslimane, D., Dustdar, S., and Sheth, A. Services mashups: The new generation of web applications. *IEEE Internet Computing 12* (September 2008), 13–15.
6. Boehm, B., and Turner, R. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003.
7. Borchers, J. *A Pattern Approach to Interaction Design*. Wiley, 2001.
8. Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M., and Grigoreanu, V. End-user mashup programming: through the design lens. In *Proc. of CHI 2010* (2010).
9. Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D., and Francalanci, C. Dashmash: a mashup environment for end user development. In *Proc. of ICWE 2011* (2011).
10. Capuano, N., Pierri, A., Colace, F., Gaeta, M., and Mangione, G. R. A mash-up authoring tool for e-learning based on pedagogical templates. In *Proc. of MTDL 2009* (2009).
11. Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., and Turransky, A., Eds. *Watch what I do: programming by demonstration*. The MIT Press, 1993.
12. Daniel, F., Casati, F., Benatallah, B., and Shan, M.-C. Hosted universal composition: Models, languages and infrastructure in mashart. In *Proc. of ER 2009* (2009).

13. Daniel, F., Koschmider, A., Nestler, T., Roy, M., and Namoun, A. Toward process mashups: key ingredients and open research challenges. In *Proc. of IWoWAaSM '09/'10* (2010).

14. Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., and Saint-Paul, R. Understanding ui integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing 11* (May 2007), 59–66.

15. Dewan, P., Agarwal, P., Shroff, G., and Hegde, R. Mixed-focus collaboration without compromising individual or group work. In *Proc. of EICS 2010* (2010).

16. Eisenstadt, M. Does elearning have to be so awful? (time to mashup or shutup). In *Proc. of ICALT 2007* (2007).

17. Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. Meta-design: a manifesto for end-user development. *Commun. ACM 47* (2004), 33–37.

18. Fischer, T., Bakalov, F., and Nauerz, A. An overview of current approaches to mashup generation. In *Proc. of WM 2009* (2009).

19. Fowler, M., and Parsons, R. *Domain-specific languages*. Addison-Wesley, 2010.

20. Fox, R., Cooley, J., and Hauswirth, M. Collaborative development of trusted mashups. In *Proc. of iiWAS 2010* (2010).

21. Goble, C., and Stevens, R. State of the nation in data integration for bioinformatics. *J. of Biomedical Informatics 41* (2008), 687–693.

22. Grammel, L., and Storey, M.-A. An end user perspective on mashup makers. Tech. Rep. DCS-324-IR, University of Victoria, September 2008.

23. Hartmann, B., Wu, L., Collins, K., and Klemmer, S. R. Programming by a sample: rapidly creating web applications with d.mix. In *Proc. of UIST 2007* (2007).

24. Hirsch, C., Hosking, J., and Grundy, J. Vikibuilder: end-user specification and generation of visual wikis. In *Proc. of ASE 2010* (2010).

25. Hoyer, V., Gilles, F., Janner, T., and Stanoevska-Slabeva, K. Sap research rooftop marketplace: Putting a face on service-oriented architectures. In *Proc. of SERVICES 2009* (2009).

26. Hoyer, V., Stanoesvka-Slabeva, K., Janner, T., and Schroth, C. Enterprise mashups: Design principles towards the long tail of user needs. In *Proc. of SCC 2008* (2008).

27. Huynh, D., Mazzocchi, S., and Karger, D. Piggy bank: Experience the semantic web inside your web browser. *Web Semant. 5* (2007), 16–27.

28. Jeffries, R., and Rosenberg, J. Comparing a form-based and a language-based user interface for instructing a mail program. *SIGCHI Bull. 18* (1986), 261–266.

29. Jhingran, A. Enterprise information mashups: integrating information, simply. In *Proc. of VLDB 2006* (2006).

30. Jones, M. C., Churchill, E. F., and Twidale, M. B. Mashing up visual languages and web mash-ups. In *Proc. of VL/HCC 2008* (2008).

31. Korfhage, R. R., and Korfhage, M. A. Criteria for iconic languages. In *Visual languages*, Plenum Press (1986), 207–231.

32. Koschmider, A., Torres, V., and Pelechano, V. Elucidating the mashup hype: Definition, challenges, methodical guide and tools for mashups. In *Proc. of MEM 2009* (2009).

33. Krahn, R., Ingalls, D., Hirschfeld, R., Lincke, J., and Palacz, K. Lively wiki a development environment for creating and sharing active web content. In *Proc. of WikiSym 2009* (2009).

34. MacLean, A., Carter, K., Lövstrand, L., and Moran, T. User-tailorable systems: pressing the issues with buttons. In *Proc. of CHI 1990* (1990).

35. Maraikar, Z., Lazovik, A., and Arbab, F. Building mashups for the enterprise with sabre. In *Proc. of ISOC 2008* (2008).

36. Maximilien, E. M., Wilkinson, H., Desai, N., and Tai, S. A domain-specific language for web apis and services mashups. In *Proc. of ICSOC 2007* (2007).

37. Myers, B. A. Evaluation of visual programming and program visualization. In *Proc. of CHI 1989* (1989).

38. Myers, B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages Computing 1* (1990), 97–123.

39. Myers, B. A., Ko, A. J., and Burnett, M. M. Invited research overview: end-user programming. In *Proc. of CHI EA 2006* (2006).

40. Namoun, A., Nestler, T., and Angeli, A. D. Service composition for non-programmers: Prospects, problems, and design recommendations. In *Proc. of ECOWS 2010* (2010).

41. Nardi, B. A. *A small matter of programming: perspectives on end user computing*. MIT Press, Cambridge, MA, USA, 1993.

42. Nestler, T., Feldmann, M., Hubsch, G., Preussner, A., and Jugel, U. The ServFace builder - a wysiwyg approach for building service-based applications. In *Proc. of ICWE 2010* (2010).

43. Nowak, M., and Pautasso, C. Goals, questions and metrics for architectural decision models. In *Proc. of SHARK 2011* (2011).

44. Ogrinz, M. *Mashup Patterns: Designs and Examples for the Modern Enterprise*. Addison-Wesley Professional, 2009.

45. Pautasso, C. Composing RESTful services with JOpera. In *Proc. of the International Conference on Software Composition (SC 2009)*, vol. 5634 of *LNCS*. Springer, 2009.

46. Soi, S., and Baez, M. Domain-specific mashups: from all to all you need. In *Proc. of Current trends in web engineering* (2010).

47. Tanimoto, S. L. Viva: A visual language for image processing. *J. Vis. Lang. Comput. 1*, 2 (1990), 127–139.

48. Thomas, J. C., and Gould, J. D. A psychological study of query by example. In *Proc. of AFIPS 1975* (1975).

49. Tuchinda, R., Szekely, P., and Knoblock, C. A. Building mashups by example. In *Proc. of IUI 2008* (2008).

50. Vrieze, P. d., Xu, L., Bouguettaya, A., Yang, J., and Chen, J. Process-oriented enterprise mashups. In *Proc. of GPC 2009* (2009).

51. Wenger, E. *Communities of practice: learning, meaning, and identity*. Cambridge University Press, 1998.

52. Wright, S. D., et al. Designing mashups with excel and visio. In *Expert SharePoint 2010 Practices*. Apress, 2011, 513–539.