A Quality Model for Mashup Components

Cinzia Cappiello[1], Florian Daniel[2], and Maristella Matera[1]

[1] DEI - Politecnico di Milano
Via Ponzio 34/5, 20133 Milano, Italy
{cappiell,matera}@elet.polimi.it
[2] University of Trento
Via Sommarive 14, 38100 Povo (TN), Italy
daniel@disi.unitn.it

Abstract. Through web mashups, web designers with even little programming skills have the opportunity to develop advanced applications by leveraging components accessible over the Web and offered by a multitude of service providers. So far, however, component selection has been merely based on functional requirements only, without considering the quality of the components and that of the final mashup. The quality in this context results from different factors, such as the software API, the contents, and the user interface.

In the literature, quality criteria for the different aspects have been proposed and analyzed, but the adaptability and dynamicity that characterize the mashup ecosystem require a separate and focused analysis. In this paper, we analyze the quality properties of mashup components (APIs), the building blocks of any mashup application, and define a quality model, which we claim represents a valuable instrument in the hands of both component developers and mashup composers.

1 Introduction

Modern Web 2.0 applications are characterized by a high user involvement: users are supported in the creation of contents and annotations, but also in the "composition" of applications starting from contents and functions that are provided by third parties. This last phenomenon is known as *Web mashups*, and is gaining popularity even under users with only little programming skills.

Mashups integrate heterogeneous *components* available on the Web, such as RSS/Atom feeds, Web services, content wrapped from third party web sites, or programmable APIs (e.g., Google Maps). Components may have a proper user interface that can be reused to build the interface of the composite application, they may provide computing support, or they may just act as plain data sources. Several mashup tools currently support the easy mashup of components, by offering visual environments where users can select pre-defined components and combine them by specifying models that abstract from technology and implementation details.

The success of a mashup is certainly influenced by the added value that the final combination of components is able to provide. However, it is self-evident

M. Gaedke, M. Grossniklaus, and O. Díaz (Eds.): ICWE 2009, LNCS 5648, pp. 236–250, 2009.

that the quality of the final combination is strongly influenced by the quality of each single component, especially if we consider the current nature of mashups: single pages where, apart from the choreography logics, the overall functionality and application behavior directly derive from the single components.

If we look at components as standalone modules, then we can say that their quality is determined by the attributes that traditionally characterize software quality. A selection and/or specialization of such attributes is however needed to capture the peculiarities deriving from the components' intended use, i.e., their combination within mashups. This factor leads us to consider components as black boxes exposing their programmatic interfaces (APIs) to the audience of mashup developers.

We strongly believe that, as for any other software product, the component-internal quality is a relevant issue, and as such it must be taken into account during component development. Nevertheless, we argue that, as also confirmed by an experimental analysis conducted on the huge set of APIs published in the programmableweb.com repository (`http://www.programmableweb.com`), for the purpose of mashup composition some external features strongly affect the component success and diffusion.

In the light of the previous observations, in this paper we discuss the quality of mashups based on a *component-driven* approach. We recognize the validity of consolidated models and metrics for the component-internal quality. Our novel contribution is a further quality perspective, which is especially oriented toward the production of successful components. More specifically, we look at *mashup components* and their *APIs* in an isolated fashion and identify those individual features (e.g., the documentation, the ease of use of the API, the content provided through the API, and so on) that are likely to contribute to the success of a component. The challenge lies in the identification of those dimensions that really affect the adoption of an API.

The paper is organized as follows. In the next section, we provide the necessary context of the paper, i.e., we describe the typical mashup scenario. In Section 3 we introduce the ISO standard for software quality, one of the starting points of our work, and we also discuss some related works. In Section 4 we look at the mashup scenario from a quality perspective and provide our own quality model for mashup components. In Section 5 we report on our first experiments, and in Section 5 we conclude the paper with a final discussion and an outlook over our future work.

2 The Mashup Development Scenario

In order to clarify the roles and artifacts we will be referring to in this paper, in Figure 1 we illustrate the typical mashup scenario that spans from the production of single *mashup components* to the integration of components into a final *mashup application*. We explicitly highlight the involved *actors* and some of the development challenges.

The *Component Developer* who wants to create a new component has to cope with two complementary concerns, i.e., functional and non functional

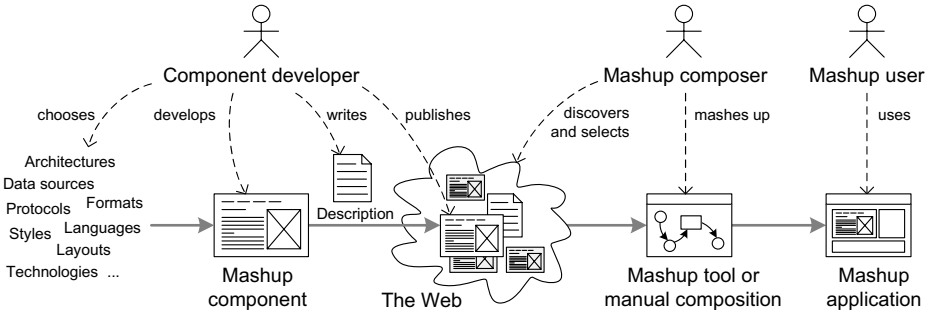


Fig. 1. The scenario for mashup API development

requirements. In this paper we concentrate on the non functional aspects and trust that the developer correctly implements all necessary functionalities.

From a non functional perspective, building a component implies taking decisions regarding the architectural style (e.g., SOAP service vs. RESTful service vs. UI component), the programming language (e.g., client side vs. server side technologies), the data formatting logic (e.g., XML vs. JSON), and so on. In addition to the functional features, all these aspects affect the "appeal" of the component from the point of view of the mashup composer that wants to include the component into a mashup application. The component developer should therefore aim at maximizing, among others, the components's interoperability, ease of use, attractiveness, and so on. Hence, based on the above considerations, the developer builds the component, provides a description or a documentation for its use (at least, ideally), and then publishes the components and its description on the Web (if any).

The *mashup composer* integrates the component into a mashup application. He typically browses the Web in search for components that suit his mashup idea, both in terms of functionality and quality provided. That is, the composer discovers components and selects the "good" ones. In doing this, he may take into account not only his own needs (e.g., a simple programming API and simple data formats for easy integration), but typically he also tries to guess the needs and the expectation of the final mashup user. Of course, a composer only selects components that will also be appealing to the users of the final mashup.

Developing good mashup components is therefore a challenging task, that requires the component developer to take into account the expectations of both the potential mashup composers and the potential mashup users. We say "potential", as it is typically not easy to fully predict who the real consumers of a component will be, once it is published on the Web. The challenge we focus on in this paper is therefore to understand how to assess the quality of a mashup component and, therefore, how to develop high-quality components.

In the rest of the paper, we assume that a *mashup component* is the logical entity that a component developer provides to the mashup composer. Physically, the component is accessed via proper *APIs*, i.e., programming interfaces that are characterized for instance by a programming language, a data format,

and a communication protocol. A single component might come with multiple APIs. For instance, a component might be used via both a RESTful API or a SOAP/WSDL API.

3 Rationale and Background

A *quality model* consists of a selection of quality characteristics that are relevant for a given class of software applications and/or for a given assessment process. Quality models are drivers of quality assessment: assessment methods relying on well-defined quality models have the merit of establishing systematic frameworks in which the different quality dimensions are identified, precisely decomposed into quantifiable attributes, and then properly measured [1].

A relevant contribution to the definition of quality models comes from a family of ISO/IEC standards that focus on the quality of software systems and on its assessment. The standard ISO 8402-86 [2] defines quality as the "totality of features and characteristics of a software product that relate to its ability to satisfy stated or implied needs". As reported in Table 1, more concretely the standard ISO/IEC 9126-1 [3] defines quality as the combination of six characteristics that represent the attributes of a software product by which its quality can be described and evaluated. For each characteristic, the standard also specifies a set of finer-grained sub-characteristics with a granularity that fits well the principal need underlying the standard definition, i.e., quantifying the quality of software by means of metrics.

The standard ISO/IEC 9126-1 also distinguishes among different perspectives:

- *Internal Quality* is based on a *white box* model that considers the intrinsic properties of the software functionality, independently of the usage environment and the user interaction, and is measured directly on the source code and its control flow.
- *External Quality* is based on a *black box* model and is related to the behavior of the software product in a given running environment.
- *Quality in use* refers to the capability of a system to enable specified users to achieve specified goals with effectiveness, productivity, safety, and satisfaction in specified contexts of use.

Based on the above framework, several works have proposed quality models for traditional Web applications (see for example [4,5,6]). Few proposals also concentrate on modern Web 2.0 applications. For example, in [7], the authors extend the ISO 9126-1 standard, and discuss the internal quality, external quality, and quality in use of Web 2.0 applications. The authors also recognize the existence of some additional factors related to the quality of contents. This dimension is indeed central in Web 2.0, due to the increasing amount of user-authored information.

There is a lack of proposals for the quality of mashups. In a sense, this is because the quality of mashups can be mainly characterized by the external quality-in-use perspective, which is exhaustively covered by the huge research

Table 1. Definition of quality characteristics in the ISO/IEC 9126 standard [3]

Characteristics	Definition	Sub-characteristics
Functionality	A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.	Suitability, Accuracy, Interoperability, Compliance, Security.
Reliability	A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.	Maturity, Fault Tolerance, Recoverability.
Usability	A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.	Understandability, Learnability, Operability.
Efficiency	A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.	Time Behaviour, Resource Behaviour.
Maintainability	A set of attributes that bear on the effort needed to make specified modifications.	Analysability, Changeability, Stability, Testability.
Portability	A set of attributes that bear on the ability of software to be transferred from one environment to another.	Adaptability, Instalabilty, Conformance, Replaceability.

on Web application usability. We however believe that beyond quality in use, other issues that are strictly related to the quality of the individual components must be considered.

Similarly to the other works described above, our model is derived from the quality attributes defined by the ISO standard. We however add a specific perspective, which allows us to concentrate on the external quality of components, i.e., on the set of properties that affect the component's quality as perceived by the mashup composer (not necessarily the final mashup user). It is worth noting that other works focused on API quality in the more general SOA (Service-Oriented Architecture) domain, by specifically addressing the set of external factors that increase the ease of use of an API (the so-called *API usability*) [8,9,10], such as the quality of API documentation [10]. Our approach capitalizes on these contributions but tries to go beyond, since it considers a broader set of external quality factors – not only usability –, all having impact on the success of mashup components.

4 A Reference Quality Model

By definition, the publication of mashup components through APIs hides their internal complexity and, therefore, also their internal details. After a component

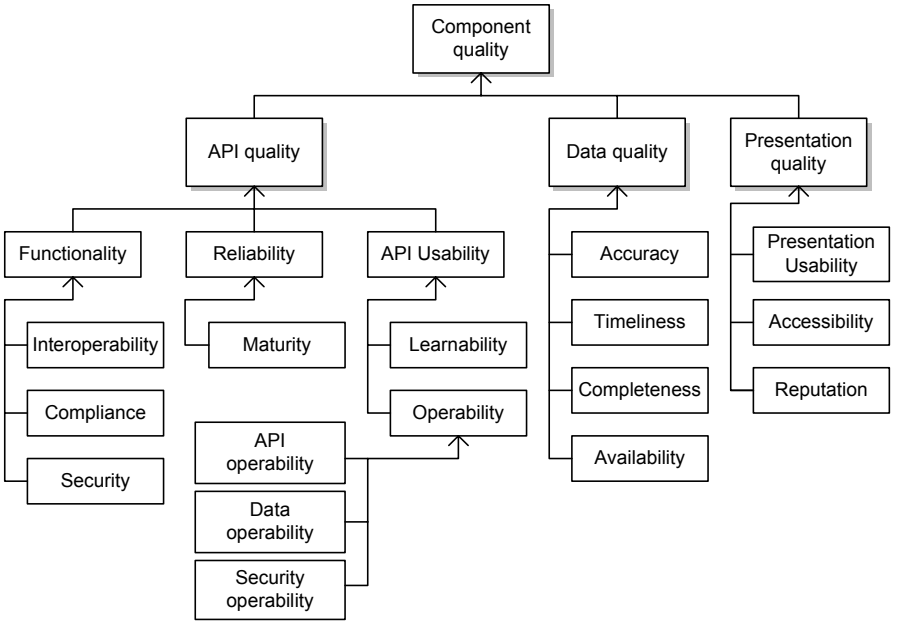


Fig. 2. The quality model for mashup components

has been deployed, external quality factors are the ones that drive the evaluation of the component's suitability for integration into a mashup application. This is also confirmed by a preliminary analysis that we have performed on the huge set of data available on programmableweb.com, a Web site that publishes data about APIs (e.g., links to the URLs for API download, descriptions, comments, user ratings, how-tos, etc.) and their use within mashups. We wrapped the data available on the site,[1] and analyzed them to identify possible correlations between observable API properties (e.g., the programming language, the number of supported protocols, the availability of documentation, etc.) and the component's usage in mashups. We discovered that the availability of *how-to* items (links to Web pages supplying information on how to install and use the component) has the strongest correlation with the diffusion of the component. This result, which is not surprising if we consider that Web 2.0 mashup composers typically prefer *easy-to-combine* components over complex components, led us to concentrate more specifically on the component-external quality.

Provided that the component-internal quality must be taken into account and must be assessed in accordance with the principles and methods traditionally adopted for software quality, in the rest of this section we concentrate on the external quality of components and illustrate our reference quality model. Figure 2 gives an overview of the addressed quality attributes, which we organize

[1] For more details on the wrapper and the analysis of the downloaded data the reader is referred to [11].

along three main dimensions, namely *API quality*, *Data quality* and *Presentation quality*, which recall the traditional "presentation-logic-data" organization of Web products. In the rest o this section we will discuss them, by highlighting the features that characterize the quality of mashup components and introducing fine-grained attributes and, where possible, assessment metrics.

4.1 API Quality

An important ingredient of the external quality of a mashup component is the set of software characteristics that can be evaluated directly on the component API. In this section, we consider three attributes that traditionally characterize the quality of software, *functionality*, *reliability*, and *usability*, revisited for the analysis of component APIs.

Functionality. Functionality can be refined by considering the interoperability, the compliance, and the security level of a component.

Interoperability is one of the most important attributes that affect the quality of a mashup component. In fact, the diffusion of a component depends on its capability to be used in different and heterogeneous environments. The interoperability of a component can be assessed by inspecting its API, since it particularly depends on the technologies used at the application and data layers. At the application layer, a mashup component can be provided through several APIs developed by using different technologies, such as different protocols or languages. The higher the number of the offered APIs for a given mashup component, the higher its interoperability. At the data layer, interoperability is affected by the number of data formats accepted for information exchange. Thus, the interoperability of a mashup component can be defined as:

$$Interoperability_{comp} = |P_{comp}| + |L_{comp}| + |DF_{comp}|$$

where $P_{comp} \subset \mathcal{P}$, $L_{comp} \subset \mathcal{L}$, and $DF_{comp} \subset \mathcal{DF}$ are the subsets of protocols, languages, and data formats used by the specific component. $\mathcal{P}$, $\mathcal{L}$, and $\mathcal{DF}$ are the sets of possible protocols, languages, and data formats that can be used for the development of mashup components. The analysis of the information contained in programmableweb.com allowed us to identify these sets. Table 2, for instance, summarizes the most prominent technologies found on programmableweb.com; the data are based on the descriptions provided by the component developers.

Table 2. Most used technologies in mashup component development

Protocols	REST, SOAP
Languages	Javascript, PHP
Data Formats	Atom, RSS, Gdata, JSON, XML, Parameter-Value

Some data formats are also standard (e.g., Atom, RSS, GData) and this increases the interoperability level and gives also the possibility to assess the *compliance* dimension as follows:

$$Compliance_{comp} = std(DF_{comp}) : DF_{comp} \rightarrow [0; 1]$$

where $std(DF_{comp})$ produces 1 when at least one of the data formats supported by the component is a standard data format, and 0 if none of the supported data formats is standard.

The *security* of a component is related to the protection mechanism that is used to rule the access to the offered functionalities. We distinguish between two aspects: *SSL support* and *authentication mechanisms*. A component might provide access to its features with or without SSL support. That is, the component might allow for encrypted communications, which improves security, or not. As for the authentication mechanism, we distinguish between no authentication, API key, developer key, and user account. If the component requires mashup composers to use an API key, this means that the composer typically needs to use an access key that is specific to the mashup application the component will be running in. The key can usually be generated on the component provider's web site (for instance, Google Maps adopts this technique). A developer key, instead, requires the mashup composer to be registered personally as developer on the web site of the component provider (eBay for instance uses this techniques), while a user account requires the mashup composer to also be a registered user of the component provider (e.g., this is necessary to integrate PayPal features into mashups). In Figure 3 we show a graphical representation of the security metric, along with two examples.

Formally, it is possible to define the security metric as

$$SEC_{comp} = SSL_{comp} + AUT_{comp}$$

where SSL_{comp} is a boolean value that indicates the use of SSL inside the component, while AUT_{comp} is a number between 1 and 4 that indicates the type of authentication method according to some complexity values, as defined in Figure 3. The score of the security metric is calculated on the basis of the actual requirements the mashup composer poses to the component. For instance, if a composer at most wants to use a developer key with SSL support, a component that imposes the use of a user account does not add any value. Instead, a component that only provides an API key or no SSL support does not meet the requirements. According to this, we assign the value that corresponds to the composer's expectation if the component meets or exceeds the expectation, and lower values to components that do not meet the expectations (see highlighted values in Figure 3).

Reliability. The black-box approach does not allow one to evaluate the level of performance of a component under stated conditions for a stated period of time. Reliability can be evaluated in terms of *maturity*, by considering the available

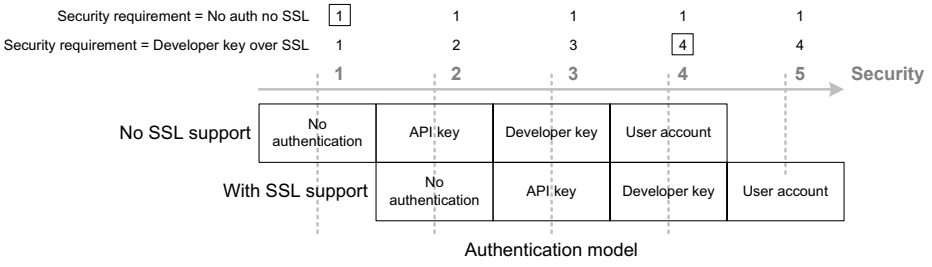


Fig. 3. Security mechanisms adopted by mashup components

statistics of usage of the component together with the frequency of its changes and updates:

$$Maturity_{comp} = max(1 - \frac{CurrentDate_{comp} - LastUseDate_{comp}}{\frac{CurrentDate_{comp} - CreationDate_{comp}}{|V_{comp}|}}; 0)$$

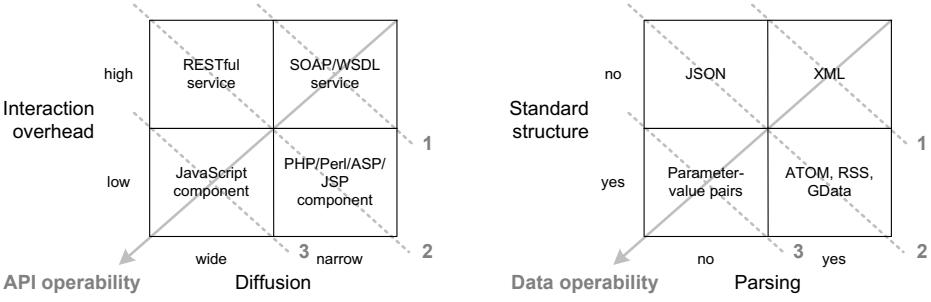
where V_{comp} is the set of versions available for a specific mashup component.

API Usability. Within the API quality dimensions, usability refers to the ease of use of the API.[2] API usability can be measured in terms of: *understandability*, *learnability*, and *operability*. Given our black box approach, *learnability* and *understandability* can be evaluated by considering the component documentation. Particularly relevant in the mashup scenario is the support offered to mashup composers by means of examples, API groups, blogs, or forums, and any other kind of documentation. The availability of each type of support contributes to increase these quality attributes.

Operability also affects the ease of use of a component. It depends on the complexity of the technologies used at the application and data layers, and of the adopted security mechanisms. The operability of technologies at the application level can be evaluated by considering the diffusion and the interaction overhead of both protocols and languages used in the API development. In fact, the diffusion of a protocol or a language enables the diffusion of a common knowledge that supports its use. In the same way, the operability of a component is higher when the interaction with the available API is easier. For example, the adoption of a protocol is more complex than the direct invocation of an object method, since dedicated standards and protocols might have to be used for the data exchange. In Figure 4(a) we show a method to estimate the operability of the most common technologies generally adopted at the application level.

Similarly, operability at the data layer can be evaluated by analyzing the data formats offered by the component along two aspects: the need for a parsing, meaning that further transformations are needed before the component can be integrated in the final mashup, and the use of a standard format. Figure 4(b) describes a method to assess the operability of the most common data formats.

[2] We will discuss presentation usability later in this section.



(a) A metric to measure operability of API types

(b) A metric to measure the operability of data formats

Fig. 4. Operability of the technologies used at the application and data level

The security operability and the actual level of security are instead inversely proportional. The higher the level of security, the lower the security operability. This is due to the consideration that operating a restrictive security solution is more demanding than less restrictive security solutions. Figure 5 represents the different degrees of security operability that can be identified by considering the security mechanisms typically adoptable in a mashup component.

In general, once the above technologies have been classified using the described criteria, it is possible to define clusters and characterize them with an operability level. As shown in Figure 4, technologies in the same cluster are associated with the same operability value. For example, in our analysis described in Figure 4, we use the following function family: $OP(T_{comp}) : T_{comp} \rightarrow OPV$, where $\mathcal{T}_{comp} = \{P_{comp}, L_{comp}, DF_{comp}, SEC_{comp}\}$ includes the technologies used by a mashup component at the application and data layers and the adopted security mechanisms, and $\mathcal{O}PV \subset \mathbb{N}$ is the set of operability values defined for each technology. Since a component can be offered by using different APIs and thus more application and data technologies have to be evaluated, the overall operability measure can be defined as:

$$OP_{comp} = max(OP(P_{comp} \cup L_{comp})) + max(OP(DF_{comp})) + max(OP(SEC_{comp}))$$

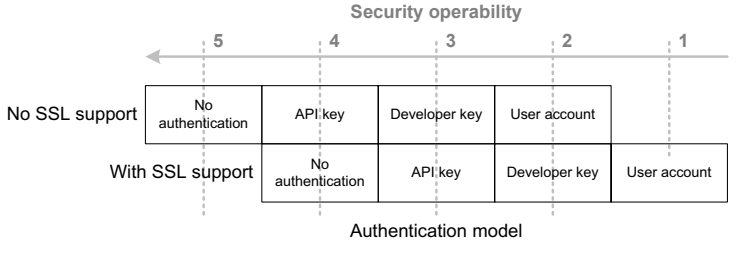


Fig. 5. Operability of the security mechanisms

The first term considers the technologies characterizing the application layer of the component; the second refers to the data layer; and the last term refers to the security mechanism implemented by the component. For each addend, we only consider the maximum operability value, as we think this characterizes best the overall operability of the component.

4.2 Data Quality

Data quality refers to the suitability of the data provided by the components through their APIs (both the information supplied to the final mashup users and the data exchanged between APIs for their choreography within the mashups). It mainly refers to data *accuracy*, *completeness*, and *timeliness*. Accuracy and completeness assess data along their correctness and numerical extension [12][13], while timeliness evaluates the validity of data along time [14]. In this context, it is also important to consider *data availability* because of data usage restrictions often applied by mashup component developers (e.g., some components limit the number of allowed requests per day).

Accuracy. It is defined as the degree with which data are consistent with the part of the real world that they have to represent. More formally, accuracy is defined as a correctness measure typically expressed in terms of proximity of a value v returned by the mashup component to a value v' considered as correct [12]. The evaluation of the accuracy dimension can be difficult if reference values are not available. In this case, digital sources can be compared, and accuracy problems are often revealed by inconsistencies among values stored in the different sources.

Completeness. It is defined as the degree with which a given data collection produced by the component includes all the expected data values. The assessment of the completeness can be performed by considering the ratio between the amount of data received and the amount of data expected:

$$Completeness = 1 - \left(\frac{Number\ of\ Missing\ values}{Total\ number\ of\ values} \right)$$

Timeliness. Represents the degree with which data are updated. It expresses how current (up-to-date) exchanged data are for the users that use them. Data can be indeed useless because they are *late* for a specific task. A measure of timeliness is defined in [14] as:

$$Timeliness = \max \left(0, 1 - \frac{currency}{volatility} \right)^{s}$$

where the exponent s controls the sensitivity of timeliness to the currency-volatility ratio. The value of the exponent is, indeed, related to the context (task-dependent), and it absorbs the subjectivity introduced with the judgment of who analyzes data.

With this definition, the value of timeliness ranges between 0 and 1, and expresses the temporal validity of data that users access. The validity is calculated by using the ratio between *currency* and *volatility*. Currency provides the "age" of data considering the creation time or the last update, while volatility is a static dimension that expresses the average period of validity of data in a specific context [14]. Temporal valid data are those data that are not "expired" when users read them.

Availability. In the SOA domain, a general assumption is to increase as much as possible the level of availability. A common practice in the definition of usage licenses for mashup components is to introduce some form of limitations. For example Google maps allows each IP up to 50,000 geocode requests per day. If, from the user perspective, this can be considered a restriction, it can be a necessary action to prevent service abuses leading to service availability pitfalls. A trade off solution must be carefully designed, so as to maximize possible quality advantages, without reducing the component's attractiveness.

4.3 Presentation Quality

Presentation quality refers to all those attributes that characterize the user experience and therefore relate to the user interface aspects that the mashup users go through when they access and use the final mashup application. It especially applies to UI components, i.e., those components that, differently form pure web services, are also provided with a presentation layer.

For this dimension, we focus on three quality attributes, i.e., *usability*, *accessibility*, and *reputation*.

Presentation usability. In some cases, mashup components are provided with a presentation layer, i.e., a user interface (UI) where some widgets provide a visualization for the component produced data and also allow some form of interaction. Despite the simplicity of such UIs, usability of the presentation mechanisms must be taken into account. All the usability attributes and metrics already defined for Web UIs can be taken into account [15,4,16]. Particular emphasis must be devoted to factors such as the *understandability* and *learnabilty*, i.e., the provision of easy-to-understand presentations for data and easy-to-learn interaction mechanisms, and the *compliance* with standard interaction mechanisms. The *attractiveness* of presentations also needs to be addressed. With this respect, RIA interfaces can provide suitable solutions.

Accessibility. All the features supporting the access by any class of users and technology must be addressed. The component UI should be therefore designed by taking into account well-know accessibility criteria, such as those defined by the W3C Web Accessibility Initiative (WAI) [17]. Just to mention few, different APIs enabling different presentation modalities should be provided for the same components, so that its contents and functions can be rendered on devices with different capabilities. Multimedia contents should be augmented with textual descriptions, so that they can be presented even through alternative browsing technologies, such as screen readers assisting impaired users. Finally, components and

the resulting mashups should be accessible through different types of hardware devices, from voice-based devices to small-size or black and white screens.

Reputation. Reputation is the degree with which a component is perceived as reliable. In the Web, most of the user actions are driven by reputation: users simply access and trust the information provided by reliable institutions and/or authors. In the mashup scenario, this trend is observable as well. Our analysis of the programmableweb.com data revealed that the most diffused components are those distributed by well-known, and therefore credible, providers (e.g., Google). Therefore, in the quality evaluation of a mashup component the credibility of the organization/person that publishes and advertises it cannot be neglected.

Form the component developer perspective, it is also important to achieve a reasonable level of reputation. Certainly, reputation is positively affected by the component documentation, especially if it is available in different formats and distributed through different channels (including blogs, forums, wikies, etc.), by the compliance of presentation mechanisms with the most diffused standards, and in general by the attitude to maximize all the quality attributes previously discussed, to meet the user (both mashup composer and mashup user) expectations.

5 Discussion

The current mashup ecosystem is characterized by a strong growth, by a strong focus on technologies, by few really value-adding mashups, and by a generally low quality of both components and mashups. The ecosystem is still in its infancy, yet the trend toward so-called "enterprise mashups" (as, for example, those supported by companies like IBM or JackBe), which go beyond 1-page Web user mashups, is real. Understanding which factors determine or influence the quality of mashups and – of particular interest to this paper – of mashup components represents a first step toward valuable mashups.

As illustrated in the scenario at the beginning of this paper, developing good, i.e., high-quality, mashup components is not a trivial task. Besides the pure functional features of a component, there are many design decisions (e.g., regarding programming languages, communication protocols, data formats, and the like) that need to be taken and that influence the quality and the success of a component. Developing a mashup component requires the component developer to take into account at least two different stakeholders, i.e., the mashup composer, who might want to include the component into his mashup, and the mashup user, who will use the component in the mashup. This is peculiar, and differentiates mashup component development from traditional development: developers of conventional APIs (e.g., Web services or object libraries) typically only need to take into account the need of developers who will use their API, as the APIs do not expose an interface that is directly operated by users; developers of Web applications, instead, rather need to take into account the users of their application, as a Web application is typically not accessed also via an API. Mashup component development, instead, must take into account the expectations of both and, hence, design decisions are harder.

In this paper, we looked at component development from an external perspective, that is, from the perspective of the mashup composer or the mashup user, and we characterized the observable properties of components in terms of a component-specific quality model. The model is based on both our own experience with the development of components and mashups, and experimental evidence gathered by analyzing data from programmableweb.com. For the actual assessment of the quality properties, we provided – where possible – metrics.

We claim that the defined model and metrics contain valuable knowledge that (i) creates an awareness of the problem of today's general low-quality in mashups and mashup components, (ii) assists the mashup composer in selecting components that effectively suit his mashup needs (focusing not only on hard functional requirements), and (iii) provides the component developer with guidelines about how to take into account the needs of both the mashup composer and the mashup user. The described model can indeed be used by the component developer as a methodology for the selection of appropriate languages, protocols, data formats, etc., compatibly with the functional requirements of the component and updated (if necessary) according to the pace of the Web 2.0.

As a next step, the model will be validated by applying it to a significant number of mashup components. We would like to "rank" mashup components (e.g., by looking at the mashups and components in programmableweb.com), in order to assess correlations among their quality properties, possibly also taking into account their use within mashups. We are also planning some formal experiments to validate our metrics against inspection-based evaluations by a pool of expert developers. We will also extend the model to cover the quality of mashups, which we believe is tightly related with the quality of the components they integrate.

References

1. Fenton, N.E., Pfleeger, S.L.: Software metrics: a rigorous and practical approach. PWS Publishing, Boston (1997)
2. ISO: ISO 8402:1994. Quality Management and Quality Assurance - Vocabulary (1986)
3. ISO/IEC: ISO/IEC 9126-1 Software Engineering. Product Quality - Part 1: Quality model (2001)
4. Calero, C., Ruiz, J., Piattini, M.: A Web Metrics Survey Using WQM. In: Koch, N., Fraternali, P., Wirsing, M. (eds.) ICWE 2004. LNCS, vol. 3140, pp. 147–160. Springer, Heidelberg (2004)
5. Malak, G., Badri, L., Badri, M., Sahraoui, H.A.: Towards a Multidimensional Model for Web-Based Applications Quality Assessment. In: Bauknecht, K., Bichler, M., Pröll, B. (eds.) EC-Web 2004. LNCS, vol. 3182, pp. 316–327. Springer, Heidelberg (2004)
6. Olsina, L., Covella, G., Rossi, G.: Web Quality. In: Web Engineering, pp. 109–142. Springer, Heidelberg (2005)
7. Olsina, L., Sassano, R., Mich, L.: Specifying Quality Requirements for the Web 2.0 Applications. In: Proc. of IWWOST 2008, pp. 56–62 (2008)

8. Ko, A.J., Myers, B.A., Aung, H.H.: Six learning barriers in end-user programming systems. In: VL/HCC, pp. 199–206. IEEE Computer Society, Los Alamitos (2004)
9. Ellis, B., Stylos, J., Myers, B.A.: The Factory Pattern in API Design: A Usability Evaluation. In: ICSE, pp. 302–312. IEEE Computer Society, Los Alamitos (2007)
10. Jeong, S.Y., Xie, Y., Beaton, J., Myers, B., Stylos, J., Ehret, R., Karstens, J., Efeoglu, A., Busse, D.K.: Improving Documentation for eSOA APIs through User Studies. In: Proc. of the Second International Symposium on End User Development (IS-EUD 2009), Siegen, Germany, March 2–4 (2009)
11. Cappiello, C.: Analyzing the Success of Mashup Components. Technical report, Politecnico di Milano (2009)
12. Redman, T.: Data Quality for the Information Age. Artech House (1996)
13. Wang, R., Strong, D.: Beyond Accuracy: What Data Quality Means to Data Consumers. Journal of Management Information Systems 12 (1996)
14. Ballou, D., Wang, R., Pazer, H., Tayi, G.: Modeling Information Manufacturing Systems to Determine Information Product Quality. Management Science 44 (1998)
15. Nielsen, J.: Web Usability. New Riders, Indianapolis (2000)
16. Matera, M., Rizzo, F., Carughi, G.T.: Web Usability: Principles and Evaluation Methods. In: Web Engineering, pp. 109–142. Springer, Heidelberg (2005)
17. Consortium, W.: Wai guidelines and techniques. Technical report (2007), `http://www.w3.org/WAI/guid-tech.html`