Information Quality in Mashups

Modern Web 2.0 applications are characterized by high user involvement: users receive support for creating content and annotations as well as "composing" applications using content and functions from third parties. This last phenomenon is known as Web mashups and is gaining popularity even with users who have few programming skills, raising a set of peculiar information quality issues. Assessing a mashup's quality, especially the information it provides, requires understanding how the mashup has been developed, how its components look alike, and how quality propagates from basic components to the final mashup application.

Cinzia Cappiello
Politecnico di Milano, Italy

Florian Daniel
University of Trento, Italy

Maristella Matera
Politecnico di Milano, Italy

Cesare Pautasso
University of Lugano, Switzerland

Mashups are applications developed by integrating content and functionality sourced from the Web. Although in most cases, enthusiastic programmers hand write them, the recent emergence of so-called mashup tools or mashup platforms, such as Yahoo Pipes (http://pipes.yahoo.com), Dapper (www.dapper.net/open/), or Intel Mash Maker (http://mashmaker.intel.com), has significantly lowered the barriers to mashup development, letting unskilled Web users easily assemble their own applications.

Mashups typically integrate heterogeneous elements available on the Web, such as RSS/Atom feeds, Web services, content scraped from third-party websites, or widgets (such as Google Maps). Different kinds of mashups reuse user interface (UI) components to build the composite application's UI, leverage and require external computational services, or simply integrate multiple plain data sources. Emerging technologies such as Web services, UI widget libraries, and tool-specific mashup (meta) models have significantly simplified access to and reuse of such building blocks, leading to a component-oriented paradigm that many current mashup platforms share.

This paradigm especially facilitates the development of so-called *situational applications*[1] — that is, applications where the developer is also the final user and that serve a highly focused purpose (for example, let you visualize apartment offers on a map) and are intended for use within a limited time horizon (until you find a suitable apartment). Situational applications typically aim to answer a precise query over a limited but heterogeneous data space. Their quality, therefore, depends strongly on the information that different integrated components can provide. Quality aspects such as maintainability, reliability, or scalability play a minor role because the final mashup is needed only for a short time. Information quality, however, is crucial for both components and composition. Assessing a mashup's quality thus requires understanding both component quality and the effect that the composition has on the final mashup's overall quality.

In this article, we introduce a quality model for mashup components and then analyze how typical composition operations affect quality (with special attention to the mashup composers' perspective). We also define a quality model for mashups (as seen from the user's perspective), with a special eye on information quality.

Quality and Mashup Development

Integrating components into a mashup typically results in a Web application. Several works have proposed quality models for Web applications,[2,3] but few proposals are specific to modern Web 2.0 applications.[4] Content quality — that is, information quality — is commonly recognized as a major factor. However, specific studies on mashup quality and on information quality's role in mashups don't yet exist.

We identify three stages in the mashup process in which information quality comes into play. Each stage has its own actor.

The *component developer* creates components for mashups. We assume that developers correctly implement the component functionality, taking into account well-known principles, best practices, and methodologies for guaranteeing the code's internal quality. From an external perspective, building a component implies making decisions about, for example, the architectural style (SOAP vs. RESTful services vs. widget APIs), the programming language (client-side such as JavaScript vs. server-side such as Ruby), the data representation (XML vs. JavaScript Object Notation), and component operability and interoperability (such as the multiplicity of APIs targeting different technologies). Such external aspects affect a component's appeal from the mashup composer's perspective.

The *mashup composer* integrates components to create a new mashup. He or she discovers components directly from the Web or from component repositories accessed from the mashup tool. Component selection takes into account each component's fitness for its purpose within the mashup and the complexity of its technological properties (for example, a simple programming API, languages, and data formats that enhance operability and interoperability), as well as the provided data's richness and completeness. The mashup composer then implements the integration logic necessary for orchestrating the components. This requires a good understanding of the components to make the most of their value and implement a high-quality mashup.

Finally, the *mashup user* isn't interested in how the mashup was built. He or she simply wants the mashup application to perform as expected, without missing data, badly aligned data, or similar information quality problems. In other words, the user is interested in the perceived external quality.

Component Quality

Publishing mashup components through APIs or services hides their internal details and gives more importance to their external properties. In line with this black-box view, in prior work,[5] we proposed a quality model for mashup components that privileges properties of the component APIs; this is indeed the perspective that's most relevant to the mashup composer or user. The model is based on both our own experience with developing components and mashups and experimental evidence gathered by analyzing data from programmableweb.com.[5,6] We organize the model along three main dimensions recalling the traditional organization of Web applications into data, application logic, and presentation layers:

- *Data quality* focuses on the suitability of the data the component has provided in terms of *accuracy*, *completeness*, *timeliness*, and *availability*.
- *API quality* refers to software characteristics that we can evaluate directly on the component API. We split API quality into *functionality*, *reliability*, and *API usability*.

Table 1. Dimensions, attributes, and subcharacteristics in the mashup component quality model.[5]

Quality dimensions	Quality attributes	Subcharacteristics
Data quality	Accuracy	Refers to data correctness and to the consistency between the data a component provides and the real-world context those data represent. Measured as the proximity of component data to correct data.
	Completeness	A component's ability to produce all expected data values. Assessed by estimating the ratio between the amount of data a component produces and the expected amount.
	Timeliness	The component output's "freshness": how up-to-date the produced data is for users. Assessed via *validity*, expressed as the ratio between currency (the data's "age" from the time of component creation or last update) and volatility (the average period of data validity in a specific context).
	Availability	Refers to possible access limitations, such as those component licenses define. Depending on the usage context, we can consider such limitations as restrictions decreasing component quality or as necessary actions to prevent abuses that can decrease component availability.
API quality	Functionality	Aggregation of API interoperability (the set of covered protocols, languages, and data formats), compliance (with respect to standard formats and technologies), and security (the provision of authentication mechanisms).
	Reliability	Corresponds to component maturity (in the API black-box approach). Assessed in terms of a component's frequency of usage and updates.
	API usability	The component API's ease of use. Measured in terms of learnability and understandability (for example, the availability of documentation, examples, blogs, or forums) and operability (the complexity of protocols, languages, data formats, and security mechanisms).
Presentation quality	Presentation usability	The usability of the presentation mechanisms adopted for interacting with UI components. Given mashups' situational nature, learnability, understandability of presentation, and compliance with presentation standards should be maximized to improve efficiency.
	Accessibility	The component presentation's ability to be "read" by any class of users and Web clients. Increases if a component offers a multiplicity of APIs supporting different presentation modalities for different devices, and also through textual annotations of multimedia content enabling alternative browsing technologies (such as screen readers assisting impaired users).
	Reputation	The component's perceived trustworthiness. Particularly affected by the component provider brand, the availability of documentation (especially if available in different formats and through different channels), and the component UI's compliance with common presentation standards.

- *Presentation quality* addresses the user experience, with attributes such as *presentation usability*, *accessibility*, and *reputation*.

Table 1 summarizes each quality attribute, also highlighting the finer-grained characteristics that refine the attribute definition.

Composition Quality

Assessing each mashup component's quality isn't enough: the final mashup application's quality also depends on how these components are interconnected. For service-based applications, the literature already provides some approaches in which quality is the main driver for service selection and composition.[7,8] Michael C. Jaeger and his colleagues assess the final applications' overall quality by aggregating the composing services' quality.[7] However, none of these approaches focuses on information quality and mashups. Mashup quality isn't simply an aggregation of individual component quality. Instead, it depends on how particular components combine into a composite logic, layout, and hence user experience.

Mashup components can be UI widgets, data

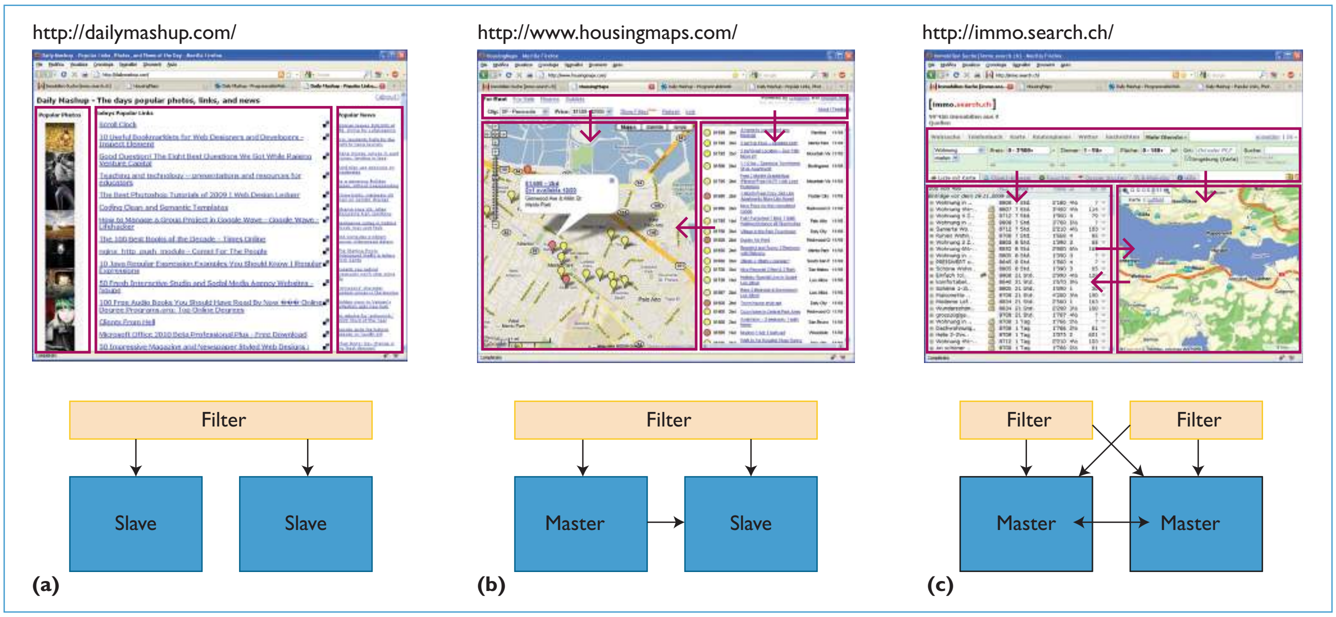

Figure 1. Basic mashup development patterns. We can see (a) the slave-slave pattern, (b) the master-slave pattern, and (c) the master-master pattern. Solid lines represent components; dashed lines represent the application integration logic.

sources, and computational services. Some are visible in the mashup, whereas others are hidden. *Hidden components* (data sources such as RSS feeds) require another component to render the data. For example, we can use an RSS reader to display the RSS feed items so that the user can inspect them and navigate through them. *Visible components* might play different roles that affect the user's perception of the final integration's quality and which we must therefore carefully take into account. By analyzing the most popular mashups published on programmableweb.com, we've identified the following typical roles:

- *Master.* Even if a mashup integrates multiple components in a single page, in most cases, one component is more important than the others. This master component is the one users interact with most. It's usually the starting point for user interaction that causes the other components to react and synchronize accordingly.
- *Slave.* A slave component's behavior depends on another component: its state is mainly modified by events originating in another (master) component. Many mashups also let users interact with slave components. However, the content items that the slave components display are selected via the user's interaction with the master component and by automatically propagating synchronization information from the master to the slaves.
- *Filter.* Filter components let users specify conditions over the content the other components show. They provide (possibly hierarchical) access mechanisms that let users incrementally select which content they want to see. Filters also reduce the size of the data sets other components show, improving the mashup's understandability and ease of use. In most cases, users specify filter conditions over the master components' data set while synchronizing slaves, so the integration logic automatically filters the slaves' content.

In short, a filter lets users select groups or sets of data items while the master component lets users select individual items that slaves will complement with additional data. Although master and slave components are usually sourced from the Web, the mashup composer develops the filter components.

Based on these three roles, our analysis of programmableweb.com mashups further lets us identify three basic patterns that characterize most mashup applications (see Figure 1) and highlights some mutual dependencies among the identified roles that impact mashup quality. (The figure shows the minimum set of components necessary for illustrating the patterns; a concrete mashup could include multiple components for the same role.).

Figure 1a shows the *slave-slave* pattern, in which the mashup integrates several slave components the user can interact with in an isolated fashion, without any propagation of data or events from one component to another. At startup or during runtime, users define filter conditions that steer all the slave components. The effect is that of a rather static application with very simple interaction facilities that lets users "query" the slave components' data set. An example is dailymashup.com, which integrates data from Flickr, Del.icio.us, furl, and Yahoo News. Regarding the resulting mashup's information quality, we assume that the filter doesn't degrade the components' perceived quality — for example, by issuing queries that theslave components can't satisfy and that would reveal data incompleteness problems. This assumption is reasonable because the mashup developer specifies the filter conditions and is aware of the selected components' coverage.

Figure 1b illustrates the *master-slave* approach, the most widely used pattern among today's mashup applications. It features all three component roles. A filter component lets users restrict the data all the other components simultaneously show. Users employ the master component to perform the main interactions with the application, such as selecting interesting data items. The slave component is automatically synchronized according to the selections performed on the master component, thereby visualizing the selected elements' details. The housingmaps.com application is a good example of a master-slave mashup: a header bar acts as the filter, letting users specify some conditions for an apartment search (such as city and price); the Craigslist table acts as master, showing the list of retrieved apartments with a link to a page with major details; and the Google map acts as the slave, showing selected apartments' locations. With the master-slave pattern, the final application's information quality could depend on the application's composition logic. Provided that master and slave are compatible in terms of data to be visualized, their integration might degrade the slave's quality. If the master provides access only to a subset of the slave data, it might prevent the user from accessing the full data the slave provides. If, instead, the master contains a superset of the slave data, it lets users ask for data items that the slave can't provide, thus revealing the slave's incompleteness.

Figure 1c shows the *master-master* pattern. This is the most complete pattern, in which — in addition to suitable filter components — all integrated components are masters. All components provide interaction facilities that let users perform selections or that provide inputs that propagate to all the other components that synchronize accordingly. The master components therefore also act as slaves. An example is the immo.search.ch application, in which — in addition to locating a housing offer on a map — moving the map lets users filter the housing offers. From an information quality perspective, the master-master pattern is similar to the master-slave pattern. If the components have different underlying data sets, situations could occur in which one component satisfies the user request, while another component can't, lowering the mashup's overall perceived quality. The master-master pattern is, however, more problematic than the master-slave pattern because it supports all directions of communication and thus increases the likelihood of revealing incompleteness problems in any of the components.

The three mashup patterns raise integration issues at the data, process, and presentation levels.[9] Integration at the process level requires setting up the necessary synchronization/orchestration logic among components using the operations and events they expose. Integration at the presentation level requires designing a composite layout, in which components are visually effective and the different presentation styles are aligned. In this article, we assume that the mashup composition performs integration at the process and presentation levels correctly. To characterize information quality in the context of mashups, we instead focus our attention on the data level.

Mashup Information Quality

Integration at the data level concerns data mediation[10] and integration.[9] The main challenge is integrating data extracted from heterogeneous sources whose exact characteristics aren't known a priori. Data integration in mashups corresponds to a *global-as-view* (GAV) problem,[11] in which the global schema is expressed in terms of views over the integrated data sources. During mashup development, the designer can inspect the attributes the components expose (the local schemas), as specified in the component APIs, and infer join attributes on which to base data integration. The underlying data instances' unpredictability, however, raises new issues, which the mashup designer can't exhaustively manage via traditional rules for integrating structured and unstructured data.[12,13]

We can characterize data integration for mashups as follows:

- Mashup applications are developed to let users retrieve and access a set of data that we call the *ideal data set* (*IDS*).
- Each component k has its own data set DS_k. To fulfill the mashup requirements, a smaller portion $SDS_k \subseteq DS_k$ could be sufficient. SDS_k is the corresponding components' *situational data set*.
- The integration of all situational data sets SDS_k gives the *real data set RDS* $\subseteq$ *IDS* that the mashup provides. *RDS*'s information quality thus depends on the quality of the data individual components provide.
- We can determine the mashup's information quality by comparing its *RDS* with the corresponding *IDS*.

Evaluating information quality in mashups requires looking at both components and composition patterns. Analogously to the data quality attributes we already defined for components, we characterize mashups' information quality by means of accuracy, completeness, timeliness, and availability. Additionally, given that integrating different data sets might lead to inconsistencies, we propose *consistency* as a new quality attribute.

We next examine each of these dimensions for the master-slave and master-master patterns; we omit the slave-slave pattern because its simple integration logic lets us express mashup quality only as an aggregation (minimum, average, maximum, or similar) of its component qualities. We further omit the filter component because we can consider the filter an auxiliary element in the composition logic whose content stems from the master component it filters. Taking into account all master components thus includes the respective filter components in the quality assessment. Finally, because we assume components are sourced from the Web, we also assume they're independent of each other. Figure 2 illustrates the situation for the master-slave pattern.

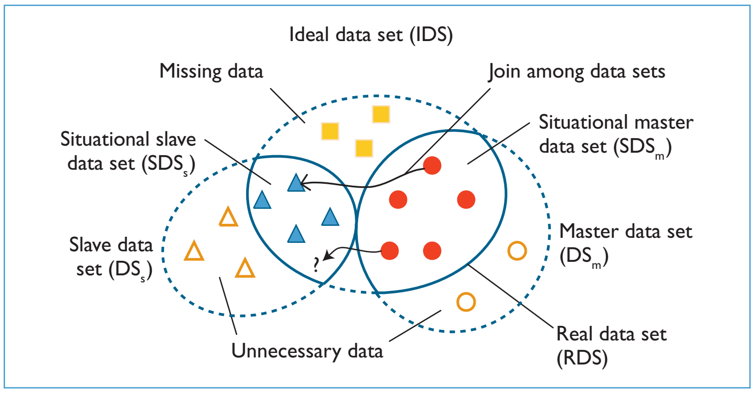

Figure 2. Data sets involved in master-slave patterns. Slave-slave patterns don't involve real data integration, whereas the master-master pattern can be seen as the composition of two independent master-slave patterns.

Accuracy

We can express a component's accuracy as the probability that its data are correct:

$$p(corr_k) = 1 - p(e_k),$$

where $p(e_k)$ is the probability that an error occurs. Data incorrectness arises each time a data value the component has produced is different from its real-world counterpart. This can happen for different reasons, such as typos, wrong representation, or missing updates. $p(e_k)$ considers all types of errors and can, for instance, be defined on the basis of a component's usage history.

In the master-slave pattern, an error might occur in both the master and slave component. Given the dependency between the master and slave, the probability of error in the slave is conditioned by the selection performed in the master. So,

$$Acc_{ms} = 1 - (p(e_m) + p(e_s \mid corr_m)).$$

We can consider master-master compositions to be the combination of two master-slave patterns: a selection in one master causes the other master to act as a slave and vice versa. So,

$$Acc_{mm} = 1 - [\alpha(p(e_{m1}) + p(e_{m2} \mid corr_{m1})) + (1 - \alpha)(p(e_{m2}) + p(e_{m1} \mid corr_{m2}))],$$

where α is the probability for one of the two master, m_1, to act as master in the user selection.

Completeness

Situational completeness SC evaluates how well components' data sets are able to provide the desired information. We can define *SC* as the degree with which the *RDS* covers the *IDS*:

$$SC = \frac{|RDS|}{|IDS|}.$$

In the master-slave pattern, *RDS* cardinality is the sum of the cardinalities of the situational master data set and the joined situational master and slave data sets. Therefore,

$$SC_{ms} = \frac{|SDS_m| + |SDS_s \, semijoin \, SDS_m|}{|IDS|}.$$

Because we can model the master-master pattern as the combination of two master-slave patterns, *RDS* cardinality results from the sum of the cardinalities of the two situational master data sets (we assume that the master data sets don't overlap, which is reasonable in that two components typically serve two different needs). Therefore,

$$SC_{mm} = \frac{|SDS_{m_1}| + |SDS_{m_2}|}{|IDS|}.$$

Situational completeness doesn't cover a case in which both the master-slave and master-master patterns have data in the slave component that aren't accessible due to missing linkages to some master data items. Consequently, we define *compositional completeness CC* as the degree with which the mashup integration effectively covers the situational data sets.

In the master-slave pattern, compositional completeness is the ratio of the cardinality of the join among the situational data sets of master and slave to the cardinality of the master's situational data set:

$$CC_{ms} = \frac{|SDS_m \, join \, SDS_s|}{|SDS_m|}.$$

In the master-master pattern, we again use a linear combination of the two corresponding master-slave patterns, with α being the probability that the first component acts as master:

$$CC_{mm} = \alpha \frac{|SDS_{m_1} \, join \, SDS_{m_2}|}{|SDS_{m_1}|} + (1-\alpha) \frac{|SDS_{m_2} \, join \, SDS_{m_1}|}{|SDS_{m_2}|}.$$

Accuracy and completeness are the quality dimensions that depend more on the choice of mashup pattern. As we will show in the next sections, the measures defined for the other dimensions are barely influenced by the type of composition.

Timeliness

Timeliness provides information about the available data sets' freshness. We can compute a mashup's timeliness as an aggregation of the individual situational data sets' timeliness values:

$Time = f_{agg}\,(time_1, \ldots\, time_k)$, where f_{agg} can be *minimum*, *average*, or *maximum*.

The timeliness evaluation is independent of the mashup patterns; the chosen aggregation function might depend on the role time plays in the application domain. For instance, considering a mashup that shows news from different newspapers, the maximum might be appropriate because it reflects the latest update. For a mashup that provides stock values for online trading, the minimum might be suitable to describe the freshness of the overall data published. If time isn't a major concern — for instance, if the mashup shows pictures on a map — the average could be a good choice.

Availability

Availability is the likelihood that the mashup can provide any data — that is, for a mashup to be available, it suffices that one of its components is available. So, we can express a mashup's availability as $Avail = 1 - \Pi_k\,(1 - Avail_k)$, where $Avail_k$ is the availability of the component k's situational data set.

Also, availability is independent of mashup patterns. However, especially in the master-slave pattern, the master's unavailability might affect the mashup's overall functionality (for instance, the user might not be able to access data in the slave), whereas the other two patterns don't present this dependency.

Consistency

Our component model assumes that each component provides consistent data — that is, components aren't contradicting themselves. If mashed up, however, situational data sets might conflict with each other, leading to inconsistency in the data shown in the mashup. For instance, when plotting university locations on a map, the map component might not be able to parse a university's address correctly and might place it on the map incorrectly (for instance, MIT might be mapped to Cambridge in the UK). Traditionally, mashup composers assess and enforce consistency through business rules expressing domain knowledge. In mashups, the composer doesn't have sufficient knowledge about the data the components provide and is, therefore, unable to write such rules in advance; thus inconsistencies only emerge during mashup execution.

When composers develop mashups as a comparison tool of multiple data sources from different providers (for instance, news feeds, as in slashdigg.com or doggdot.us), inconsistency might not be problematic because it's up to users to compare via the mashup the results of querying different data sources and infer which one they should trust as the one providing the most correct and timely data.

Information quality is highly relevant in mashup development: a mashup's quality is sensitive to both its components' quality and the way components are integrated. Whereas components with low information quality can't lead to a high-quality mashup, the composition logic could introduce additional quality issues (such as inconsistencies). Developing high-quality mashups turns out to be nontrivial, and mashup composers should be assisted in their task.

In the future, we plan to look into collaborative (wiki-style) mashups and user-driven inconsistency resolution techniques, where users might be able to influence the quality of the information the mashup presents by providing feedback (ranking, weighing, or correcting information items, for example). This will require extending mashup architectures to store and manage user feedback — a significant departure from mainstream mashup architectures, which has interesting and unexplored implications on mashup quality.

References

1. A. Jhingran, "Enterprise Information Mashups: Integrating Information, Simply," *Proc. Very Large Databases Conf.* (VLDB 06), ACM Press, 2006, pp. 3–4.
2. C. Calero, J. Ruiz, and M. Piattini, "A Web Metrics Survey Using WQM," *Proc. Int'l Conf. Web Eng.* (ICWE 04), Springer, 2004, pp. 147–160.
3. L. Olsina, G. Covella, and G. Rossi, "Web Quality," *Web Engineering: Theory and Practice of Metrics and Measurement for Web Development*, E. Mendes and N. Mosley, eds., Spinger-Verlag, 2005, pp. 109–142.
4. L. Olsina, R. Sassano, and L. Mich, "Specifying Quality Requirements for the Web 2.0 Applications," *Proc. 7th Int'l Workshop on Web-Oriented Software Technologies*, (IWWOST 08), Vydavatestvo STU, 2008, pp. 56–62.
5. C. Cappiello, F. Daniel, and M. Matera, "A Quality Model for Mashup Components," *Proc. Int'l Conf. Web Eng.* (ICWE 09), Springer, 2009, pp. 236–250.
6. S. Yu and C.J. Woodard, "Innovation in the Programmable Web: Characterizing the Mashup Ecosystem," *Proc. Int'l Conf. Service Oriented Computing* (ICSOC 08), Springer, 2008, pp. 136–147.
7. M.C. Jaeger, G. Rojec-Goldmann, and G. Muhl, "QoS Aggregation in Web Service Compositions," *Proc. IEEE Int'l Conf. e-Technology, e-Commerce, and e-Service* (EEE 05), IEEE CS Press, 2005, pp. 181–185.
8. Q. Liang, X. Wu, and H.C. Lau, "Optimizing Service Systems Based on Application-Level QoS," *IEEE Trans. Services Computing*, vol. 2, no. 2, 2009, pp. 108–121.
9. G. Di Lorenzo et al., "Data Integration in Mashups," *SIGMOD Record*, vol. 38, no. 1, 2009, pp. 59–66.
10. G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *Computer*, Mar. 1992, pp. 38–49.
11. M. Lenzerini, "Data Integration: A Theoretical Perspective," *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems* (PODS 02), ACM Press, 2002, pp. 233–246.
12. A. Motro and P. Anokhin, "Fusionplex: Resolution of Data Inconsistencies in the Data Integration of Heterogeneous Information Sources," *Information Fusion*, vol. 7, no. 2, 2006, pp. 176–196.
13. F. Naumann, J. Christoph Freytag, and U. Leser, "Completeness of Integrated Information Sources," *Information Systems*, vol. 29, no. 7, 2004, pp. 583–615.

Cinzia Cappiello is an assistant professor at the Politecnico di Milano. Her research interests regard data and information quality aspects in service-based and Web applications, Web services, and sensor data management. Cappiello has a PhD in information technology from Politecnico di Milano. She cochairs several workshops and regularly serves as reviewer for international conferences and journals in the data and information quality area. Contact her at cappiell@elet.polimi.it; http://ho me.dei.polimi.it/cappiell.

Florian Daniel is a postdoctoral researcher at the University of Trento, Italy. His main research interests are mashups and user interface composition approaches for the Web, Web engineering, quality, and privacy in business intelligence applications. Daniel has a PhD in information technology from Politecnico di Milano, Italy. He's coauthor of the book *Engineering Web Applications* (Springer, 2009) and co-organizer of the International Workshop on Lightweight Integration on the Web (ComposableWeb). Contact him at daniel@disi.unitn.it; www.floriandaniel.it.

Maristella Matera is assistant professor at Politecnico di Milano. Her current research interests span Web mashups, Web engineering models and design methods, quality in Web engineering, Web adaptivity, and context-awareness. Matera has a PhD in information technology from Politecnico di Milano. She's author of roughly 100 papers and is coauthor of the books *Designing Data-Intensive Web Applications* (Morgan Kaufmann, 2002) and *Engineering Web Applications* (Springer, 2009). Contact her at matera@elet.polimi.it; http://home.dei.polimi.it/matera/.

Cesare Pautasso is an assistant professor in the Faculty of Informatics at the University of Lugano, Switzerland. His research in the area of software architecture currently focuses on building experimental systems for Web 2.0 mashups and RESTful service composition by means of business process modeling languages. Pautasso has a PhD in computer science from ETH Zurich. He's the lead architect of JOpera, a powerful rapid service composition tool for Eclipse. He's a member of the IEEE Computer Society, the ACM, and the Swiss Association for Research in Information Technology. Contact him at c.pautasso@ieee.org; www.pautasso.info and follow him on twitter.com/pautasso.