Mashups have emerged as an innovative software trend that re-interprets existing Web building blocks and leverages the composition of individual components in novel, value-adding ways. Additional appeal also derives from their potential to turn non-programmers into developers. Daniel and Matera have written the first comprehensive reference work for mashups. They systematically cover the main concepts and techniques underlying mashup design and development, the synergies among the models involved at different levels of abstraction, and the way models materialize into composition paradigms and architectures of corresponding development tools. The book deliberately takes a balanced approach, combining a scientific perspective on the topic with an in-depth view on relevant technologies. To this end, the first part of the book introduces the theoretical and technological foundations for designing and developing mashups, as well as for designing tools that can aid mashup development. The second part then focuses more specifically on various aspects of mashups. It discusses a set of core component technologies, core approaches, and architectural patterns, with a particular emphasis on tool-aided mashup development exploiting model-driven architectures. Development processes for mashups are also discussed, and special attention is paid to composition paradigms for the end-user development of mashups and quality issues. Overall, the book is of interest to a wide range of readers. Students, lecturers, and researchers will find a comprehensive overview of core concepts and technological foundations for mashup implementation and composition. Even without low-level coding details, practitioners like software architects will find guidance on key implementation concepts, architectural patterns, and development tools and approaches. A related website provides additional teaching material which can be used either as part of a course or for self study. This book is timely, provides a thorough scientific investigation and also has practical relevance in the general area of composition and mashups. It is of particular interest to researchers and professionals wishing to learn about relevant concepts and techniques in service mashups, composition, and end-user programming.

From the Preface by Boualem Benatallah, University of New South Wales, Sydney

Chapter 8
Tool-Aided Mashup Development

Figures
8.3 Abstracting Components

Fig. 8.1 Wrapping components into a unified view on native component models.
The model is based on four abstractions: state, event, parameter, and operation. Each component is the process or process step that is being displayed. Model components wrap UI, application, and data services and expose them and making them available only upon explicit request from the outside via operations of the common component model. In practice, it may therefore be necessary to develop suitable wrappers that wrap different component types (or components) into a common component model that exposes common access to operation parameters. If a function returns a result, the receipt is represented as a set of name-value pairs. What the state exactly contains and its treatment and recording is of type is of type is of type. In our case study, the state for the component we are integrating in the end are always components that comply with the model. Say that what we integrate in the end are always components that comply with the model.

**Fig. 8.2** Unified component model of mashArt components for SOAP/RESTful web services, UI components and RSS/Atom feeds [90].
A mashup must contain exactly one sink.

A data flow connector has exactly one source and one target.

8.4.1 A simple example

Both to recall the basic meta-modeling concepts and to show an example that is easy to understand, in this section we develop a simple modeling language for the development of data mashups. The language is not used in any concrete mashup platform, and serves rather the didactic purpose of illustrating how to develop a mashup modeling language.

Let’s assume we want to support the development of data mashups with the following simple set of requirements:

• A mashup integrates RSS feeds only, where each feed is identified by a unique name and the URL of the feed.

• A mashup has two types of operations: the union operation allows one to merge multiple RSS feeds into one, e.g., by concatenating them; the filter operator allows one to filter out items of an RSS feed that satisfy a given condition, e.g., expressed in JavaScript or any other language.

• The end of a mashup’s integration logic is uniquely identified by a sink component, which provides for the publication of the mashup output again as an RSS feed.

• Components and operators of the mashup are connected via suitable data flow connectors.

In Figure 8.3 we draw a possible metamodel for the target mashup language. Each construct of the modeling language that a developer needs to operate (e.g., draw or provide an input for) is represented by an own concept of the model. The most interesting concept in the model is the data flow connector, which – according to our interpretation of the above requirements, expressed as a comment in the metamodel – has exactly one source and one target.

Fig. 8.3 Metamodel (M2) of a very simple data flow mashup language: it supports fetching different RSS feeds from the Web, computing their union and/or filtering them, and publishing the result again as an RSS feed on the Web (the sink). A data flow connector must always have exactly one source and one target.
Fig. 8.4 A simple data flow model (M1) complying with the metamodel of Figure 8.3 expressed in an abstract syntax, i.e., a UML object diagram.
Fig. 8.5 The simple data flow model (M1) of Figure 8.4 expressed in a concrete syntax that highlights the semantics of the constructs and eases readability.
The pipe that implements the required feature is illustrated in Figure 1. It is composed of five components: The URL Builder is needed to set up the remote GeoNames service, which takes a news RSS feed as input, analyzes its content, and inserts geo-coordinates, i.e., longitude and latitude, into each news item (where possible). Doing so requires setting some parameters: Base = http://ws.geonames.org, Path elements = rssToGeoRSS, and Query parameters = FeedUrl: news.google.com?topic=t&output=rss&ned=us. The so-created URL is fed into the Fetch Feed component, which loads the geo-enriched news feed. In order to filter out the news items we are really interested in, we need to use the Filter component, which requires the setting of proper filter conditions via the Rules input field. Feeding the filtered feed into the Location Extractor component causes Pipes to plot the news items on a Yahoo! Map. Finally, the Pipe Output component specifies the end of the pipe.

If we analyze the development steps above, we can easily understand that developing even such a simple composition is out of the reach of people without programming knowledge. Understanding which components are needed and how they are used is neither trivial nor intuitive. The URL Builder, for example, requires the setting of some complex parameters. Then, components need to be suitably connected, in order to support the data flow from one component to another, and output parameters must be mapped to input parameters. But more importantly, plotting news onto a map requires knowing that this can be done by first enriching a feed with geographical location information, then fetching the actual feed, and only then the map is ready to plot the items. Enabling non-expert developers to compose a pipe like the above requires telling (or teaching) them the necessary knowledge. In WIRE, we aim to do so by providing non-expert developers with interactive development advice or composition, inside Fig. 8.6

As implemented, and plots it on a map (the Location Extractor component).

Fig. 8.6 A simple pipe that enriches an RSS feed with geographical location information and plots it on a map (the Location Extractor component).
Fig. 8.7 A simplified metamodel of Yahoo! Pipes for the pipe in Figure 8.6.
8.4.3 mashArt

As last example of mashup metamodel, we briefly study the internals of the mashArt platform [90], which proposes an integration approach called by the authors "universal integration." Universal integration in this context refers to the integration of data, application logic and UIs inside one and the same modeling environment. Specifically, mashArt supports SOAP and RESTful Web services, RSS/Atom feeds, as well as a proprietary format of JavaScript-based UI components (similar to W3C widgets) [282].

Figure 8.8 shows an example mashup modeled in mashArt. The model represents a simple application for the monitoring of compliance.

© Copyright 2014 by F. Daniel and M. Matera. Reproduction for classroom use and teaching allowed if source is properly cited.
Fig. 8.9 Metamodel of the mashArt modeling notation based on the unified component model of Figure 8.2.
Fig. 8.10 A screenshot of Presto Wires for data mashups serialized in EMML.
The First Line Support (FLS) system is built to facilitate the FLS team of TIE with a new portal-based solution based on the features of OMELETTE. The context is given by the First Line Support scenario of D7.1 document.

The FLS team of TIE provides maintenance service and support for TIE products. The FLS team helps clients with license renewal and also with troubleshooting of TIE products. This requires regular communication and frequent information exchange. So this portal solution with facilities like integrated telco services and real-time communication helps support people to effectively communicate and exchange information.

Fig. 8.11 A screenshot of the Apache Rave mashup environment extended by the EU FP7 project OMELETTE to import/export OMDL-compliant workspaces [262].
detailed in Section 6.3, applying these constraints allows for an unambiguous translation of the meta-model into a formal - and machine-readable - language schema, which is then needed for the definition of other artifacts of the system. In addition, using this constrained modeling language also opens to future extensions of the meta-model by third parties, making them aware of the implications of each model extension or modification on the resulting language definition (since deterministic translation rules are defined). Concretely, as defined by the meta-meta-model depicted in Figure 4, the meta-model may consist of:

- **Entities**: Represent main constructs of the composition language. They are identified by a name.

- **Attributes**: Each entity can have a set of related attributes characterizing it. Attributes have a name and a type. The type can be stated through its name or can be explicitly defined in form of enumeration of possible values. To be noticed, each entity in our meta-model must contain an attribute named `id`, representing a unique identifier for the instances of the entity used to reference them.

- **Associations**: Relations among the entities are expressed through associations. Only two possible types of associations are needed: **composition** and **uni-directional association**. The composition is used to state that an entity is contained in another one, while the uni-directional association states that an entity simply refers to another entity, but it is not contained in it.

- **Cardinalities**: Represent associations' multiplicities. The target cardinality represents the multiplicity of the association when reading it following the specified association direction, while the source cardinality represents the multiplicity when reading the association in the opposite direction.

Patterns are based on a **generic mashup language model** (not a metamodel). The model does not yet represent an executable language. It syntactically puts composition constructs and features into relation with each other, but it also contains constructs and features that may not be compatible with each other (e.g., control flow and data flow paradigms). The model determines which features are supported and how they are syntactically integrated; the sensible design of feature constraints provides for soundness. Hence, given a set of non-conflicting composition features, the custom composition language is represented by the union of the respective reference specifications. Similarly, a **custom component description language** can be derived, which can be used as guide for the implementation of components or component wrappers and to describe their external interfaces. Both the custom mashup language and the custom component description language are then mapped 1:1 to XSD, so as to enable the definition of mashups in XML and the automatic checking of the conformance of mashups with the reference language model.

The approach is therefore to **compose** mashup languages out of composition features represented as language patterns. Just like in any other composition approach, the core problem is therefore the identification and formalization of the "components" to work with. Here, these components are **language patterns** (e.g., XSD fragments). However, these patterns have a distinctive feature that makes the problem very different from generic component-based

---

**Fig. 8.12** Conceptual approach to developing custom mashup languages [254].
Fig. 8.13 The generic mashup language model bringing together the most common mashup features [253].
8.6 Developing Mashup Languages

Selected language features

1. data_flow
2. service_component
3. REST_for_service
4. data_component
5. RSS_for_data
6. atom_for_data
7. min_1_operation_per_component
8. max_1_operation_per_component
9. request_response
10. min_1_input_param_per_operation
11. max_N_input_param_per_operation
12. min_1_output_param_per_operation
13. max_1_output_param_per_operation
14. manual_input
15. configuration_param
16. branch

Fig. 8.14 Yahoo! Pipes example composition and set of respective language features
Fig. 8.15 Conceptual reference architecture of a mashup platform articulated into front-end, back-end and persistent data store.
As illustrated in Figure 8.15, debugging and testing does not ask only for suitable extensions of the mashup editor: also the mashup runtime environment must come with suitable capabilities that allow the developer (i) to inspect the state of a mashup during its execution at different points in time and/or modeling constructs (for debugging) and (ii) to test mashups under development in a protected environment. Ideally, this latter feature allows the developer to run and test the functioning of a mashup under development without producing any side-effects in the mashup platform (e.g., on the persistent storage of the platform or external services). Such kind of "sandboxing" of mashup executions is hard to implement, and therefore not yet fully supported by mashup platforms (to the best of our knowledge).

A concrete example where such a mashup sandbox would be very useful are, for instance, the telco mashups discussed in Section 7.4. Telco APIs typically provide access to telco capabilities that require some form of payment, e.g., a monthly subscription or pay-per-use payments. For development, all telco operators provide developers with suitable development accounts, which usually provide limited access to APIs for free. The problem is that each telco operator has own rules, own payment options, and own authentication mechanisms. In order for a developer to be able to seamlessly test and debug a

---

Fig. 8.16 Screen shot of Yahoo! Pipes with the debugging tab open (bottom). The content of the tab is the output of the Fetch Feed component selected in the canvas.
Finally, even with specifically tailored abstractions, graphical notations, and debugging and test support mashup development may be a tricky endeavor. A good mashup platform therefore provides developers also with suitable help resources, which allow them to learn how to use the platform and how to develop mashups with the platform. The most common help resources are component descriptions, development notation explanations, tutorials, and example mashups that can be used as reference for the implementation of new mashups. Yahoo! Pipes facilitates, for example, the reuse of existing mashups by providing a dedicated mashup cloning functionality, which allows a developer to copy and paste an existing pipe model for modification and evolution into a new pipe model.

Practice has however shown that documentation is never enough and complete. This is especially true in the context of software development, which is an area that is constantly under evolution and change. The best way to
address possible lacks in the documentation is the use of developer communities, which allow the developers of a mashup platform to exchange their experience and to leverage on the whole developer community's knowledge (e.g., via discussion forums). Figure 8.17 shows, for example, a screenshot of JackBe Presto's developer community website, which interconnects developers with other developers as well as with experts by JackBe itself. Developer communities are an effective instrument for the bottom-up organization of development knowledge and can be considered an integral part of many software development communities.

Other ways to provide assistance are based on recommendation on frequently adopted components and composition patterns, that the tools generate by mining large repositories of mashup models. Chapter 9 will illustrate such assistance mechanisms.

8.7.2 Mashup execution and operation

8.7.2.1 Execution

Some details about the internals of mashups, i.e., their internal architecture, we already discussed in Chapters 6 and 7. The development of runtime environments for the execution of mashups developed with mashup tools asks for software architectures that are the result of the joint application of that knowledge and the knowledge about how to develop parameterized, generative architectures for architecture-centric, model-driven software development (AC-MDSD) [256] (see Chapter 4).

In Figure 8.18, we recall the generic architecture for code generation in AC-MDSD. In the context of mashup platforms, the core of the runtime environment corresponds to the components marked as generative architecture. The infrastructure components are, for instance, the various protocol adapters, Application Platform, Individual code, Application model, Repetitive code, and Code generator. The generative architecture uses Configuration files, which are generated by the code generator. The code generator is fed by Generation templates and generates Individual code, which is used by the Application. The Application model is connected to the Configuration files, which are used by the Repetitive code. The Infrastructure components provide the necessary support for the execution of the application.

Fig. 8.18 Code generation in architecture-centric MDSD. The generative architecture corresponds to the runtime environment in Figure 8.15.