Mashups
A Journey from Concepts and Models to the Quality of Applications

Cinzia Cappiello – Politecnico di Milano
Florian Daniel – Università di Trento
Maristella Matera – Politecnico di Milano
Learning Objectives

1. **Mashup definition and characterization**
   - Classifying dimensions, contexts of use, target users, benefits

2. **Mashup models**
   - Conceptual underpinning of mashups for different mashup types

3. **Mashup tools and composition paradigms**
   - How mashup models can materialize into platforms for assisted mashup development

4. **Mashup quality**
   - Quality issues for components and mashups, going beyond traditional quality models and practices
CORE ASPECTS AND DEFINITIONS
Technological and societal context

- From one-way communication medium (Web 1.0) to a distributed and democratic communication platform (Web 2.0)
- User-driven innovation
- SOA, SaaS, HTML5, sophisticated devices

Web mashups as innovative software to reinterpret existing building blocks by composing them in an value-adding manner
• The term mashup is widely used today
• Typical discussion points:
  • UI or not?
  • Web accessible resources or not?
  • Client-side technologies or also server-side languages?
The housingmaps.com mashup

Provides for the synchronized exploration of housing offers from craigslist.com and maps by Google Maps

Integration is the added value provided by the mashup
Mashup definition

A **mashup** is an application that integrates two or more **mashup components** at any of the **application layers** (data, application logic, presentation layer) possibly **putting them into communication** among each other.

**Mashup component**: any piece of data, application logic and/or user interface that can be reused and that is accessible either locally or remotely.

**Mashup logic**: is the internal logic of operation of a mashup; it specifies the invocation of components, the control flow, the data flow, the data transformations, and the UI of the mashup.
The added value...

No added value

Additional information, functions, visualizations!
Other definitions

“Web-based resources consisting of dynamic networks of interacting components” (Abiteboul et Al., 2008)

“API enablers” (Ogrinz, 2009), to create an own API where there is none

“Combination of content from more than one source into an integrated experience” (Yee, 2008)
However...

- There are many applications that would not qualify as mashups
- The classification does not help characterize the mashup ecosystems from an engineering perspective
Introduction

Mashup classes like Web mashups, mobile mashups, telco mashups, data mashups and similar bear some more semantics from this perspective. Yet, there still seems to be an arbitrary proliferation of prefix-mashup combinations, without any evident connection among them and, more interestingly, these types of mashups are not mutually exclusive (for example, there may be mobile telco mashups or Web data mashups). We therefore analyzed the most important mashup types we found in literature and found that the prefixes used by these classifications can almost all be fit into one of three perspectives. We graphically illustrate the resulting ecosystem of mashups and mashup definitions as a cube in Figure 1.3. The perspectives are:

- **Composition**: This perspective emphasizes the internals of a mashup, i.e., its components and how these are composed into a new application. This perspective is the one driving our own definition of mashup in Definition 1.1. It stems from the traditional separation of concerns in software development, which separates an application into three layers, a data layer, an application logic layer and a presentation layer. This separation of concerns did not influence only how applications are structured internally, but it also fostered the growth of suitable API and component types at the three layers, so as to ease interoperability and integration of the layers. Looking at which layer of the application stack a mashup is composed, the composition perspective therefore groups mashups into data mashups, logic mashups, UI mashups, and hybrid mashups (any combination of the former three types).

- **Domain**: This perspective emphasizes the purpose of a mashup, i.e., the functionality it aims to provide. Partly, the tags used by developers to describe their mashups fall into this perspective, as they too describe delimited domains. The domains in this context may be essentially of two types: technological domains, such as for telco mashups or mobile mashups, and application domains, such as for social mashups or mapping mashups.

Three different perspectives on the mashup ecosystem
Mashup positioning in relation to other integration practices

Mashups introduce integration at the presentation layer and typically focus on non-mission-critical applications.
The long tail model

Number of users

Applications

The long tail of the software market and its opportunities for mashups
Computer-assisted composition
Benefits

- Knowledge transfer from end-users to developers
- Higher satisfaction for end-users
- Reduced costs for product evaluation
- User involvement in the creation of applications
Other benefits

• Easy development of situational applications for **power users**
• Fast prototyping for **developers**
• Increased ROI for **SOA investments**
• Increased visibility by **content/component providers**
The research perspective

• Mashup development is non-trivial
  – A very large set of (heterogeneous) technologies and integration techniques
  – New technologies and interaction modalities emerge at fast pace

• Luckily, mashups typically work on the “surface”
  – Reuse of existing components - neglecting the complexity hidden behind the service's external interface
  – Composition of the outputs of (much more complex) software systems

• The work of developers can be facilitated by suitable abstractions, component technologies, development paradigms and enabling tools

• Mashup development practices are increasingly becoming the very object of scientific investigations
Part I

MASHUP MODELS
Learning Objectives

1. Introducing models for different mashup types

2. Introducing typical architectural patterns

3. Identifying the peculiarity of UI integration
Basic mashup model

A mashup integrates a set of components, possibly puts them into communication, and optionally renders results or components.
Data mashups

- Fetch data from different resources, process them, and return an integrated result set

Logic mashups

- Integrate functionality published by logic or data components

User Interface (UI) mashups

- Combine the component's native UIs into an integrated UI; the components’ UIs are possibly synchronized among each other

Hybrid mashups

- Span multiple layers of the application stack, bringing together different types of components inside one and a same application; integration happens at more than one layer
• **Data components**
  - RSS and Atom feeds, XML JSON, CSV and similar data resources, web data extractions, micro-formats, SOAP or RESTful services that are *used as data services only*

• **Logic components**
  - SOAP and RESTful web services, JavaScript APIs and libraries, device APIs, and API extractions

• **UI components**
  - Code snippets and JavaScript UI libraries, Java portlets, widgets and gadgets, web clips and extracted UI components
Mashups are a new type of integration, fueled by the constantly growing availability of reusable resources on the Web. In the previous chapter, we have seen that the Web offers a very rich set of reusable resources, which may turn mashup development into a complex endeavor. The chapter is structured according to the four values of the first mashup characteristic, the mashup type. Considering the possible values of the other characteristics gives an idea of the variety of the possible implementations of these basic types of mashups.

One interesting aspect of mashups, as applications focusing on the reuse of web resources, which we did not consider throughout our discussion, is the reusability of mashups themselves as components for other mashups. Alternatively, we can also speak about hierarchical composition. While for data and logic mashups this is not an issue, as they typically are delivered via RSS/Atom feeds or web services that are reusable components by definition, reusability becomes an issue for UI and hybrid mashups. Of course, it is always possible to extract pieces of UI from any kind of web application, but this is generally not a good practice, and one would simply expect more attention to this problem from mashups. However, as we will see when discussing mashup platforms, the publication of mashups as reusable components is mostly neglected or hard to achieve with current implementations, e.g., based on the runtime interpretation of mashup model instances (diagrams).

As for the further reading, we already discussed data and application integration in Chapter 2, introducing the basics and foundations underlying both data and logic mashups. Doan et al. [107] provide a very good summary of the problems and solutions regarding data integration. Alonso et al. [13] surely represent a reference for application integration with a special focus on the case of web services, while Papazoglou [224] more specifically explains the use of web service technologies in practice, e.g., also providing good insight into the problem of correlation in the context of BPEL. Although mashup do not aim at the full power of BPEL, the problem represents itself in similar terms.

### Possible architectural configurations, compatible with the requirements of the chosen components

#### Client-side mashups
- e.g., UI mashups

#### Server-side mashups
- e.g., data and logic mashups

#### Client-server mashups
- e.g., hybrid mashups with user interfaces
• **UI-based integration**
  – The UI of the mashup acts as a container
  – Components run in a completely isolated fashion

• **Orchestrated integration**
  – Centralized composition logic, orchestrating component execution

• **Choreographed integration**
  – Each component participating in a choreography is individually able to send and receive messages
  – The mashup puts into place only the communication infrastructure
6.7 Summary and Bibliographic Notes

Architectural configurations that we proposed throughout this chapter. Figure 6.20 recalls the seven characteristics.

Mashups are a new type of integration, fueled by the constantly growing availability of reusable resources on the Web. In the previous chapter, we have seen that the Web offers a very rich set of reusable resources, which may turn mashup development into a complex endeavor. The chapter is structured according to the four values of the first mashup characteristic, the mashup type. Considering the possible values of the other characteristics gives an idea of the variety of the possible implementations of these basic types of mashups.

One interesting aspect of mashups, as applications focusing on the reuse of web resources, which we did not consider throughout our discussion, is the reusability of mashups themselves as components for other mashups. Alternatively, we can also speak about hierarchical composition. While for data and logic mashups this is not an issue, as they typically are delivered via RSS/Atom feeds or web services that are reusable components by definition, reusability becomes an issue for UI and hybrid mashups. Of course, it is always possible to extract pieces of UI from any kind of web application, but this is generally not a good practice, and one would simply expect more attention to this problem from mashups. However, as we will see when discussing mashup platforms, the publication of mashups as reusable components is mostly neglected or hard to achieve with current implementations, e.g., based on the runtime interpretation of mashup model instances (diagrams).

As for the further reading, we already discussed data and application integration in Chapter 2, introducing the basics and foundations underlying both data and logic mashups. Doan et al. provide a very good summary of the problems and solutions regarding data integration. Alonso et al. surely represent a reference for application integration with a special focus on the case of web services, while Papazoglou more specifically explains the use of web service technologies in practice, e.g., also providing good insight into the problem of correlation in the context of BPEL. Although mashup do not aim at the full power of BPEL, the problem re-presents itself in similar terms.

---

**Mashup characteristics**

1. Mashup type
2. Component types
3. Runtime location
4. Integration logic
5. Data passing logic
6. Presentation logic
7. Instantiation model

How components exchange data:

- Direct data passing
- Blackboard vs. Shared memory
- Mediated data passing
6.7 Summary and Bibliographic Notes

architectural configurations that we proposed throughout this chapter. Figure 6.20 recalls the seven characteristics.

Mashups are a new type of integration, fueled by the constantly growing availability of reusable resources on the Web. In the previous chapter, we have seen that the Web offers a very rich set of reusable resources, which may turn mashup development into a complex endeavor. The chapter is structured according to the four values of the first mashup characteristic, the mashup type. Considering the possible values of the other characteristics gives an idea of the variety of the possible implementations of these basic types of mashups.

One interesting aspect of mashups, as applications focusing on the reuse of web resources, which we did not consider throughout our discussion, is the reusability of mashups themselves as components for other mashups. Alternatively, we can also speak about hierarchical composition. While for data and logic mashups this is not an issue, as they typically are delivered via RSS/Atom feeds or web services that are reusable components by definition, reusability becomes an issue for UI and hybrid mashups. Of course, it is always possible to extract pieces of UI from any kind of web application, but this is generally not a good practice, and one would simply expect more attention to this problem from mashups. However, as we will see when discussing mashup platforms, the publication of mashups as reusable components is mostly neglected or hard to achieve with current implementations, e.g., based on the runtime interpretation of mashup model instances (diagrams).

As for the further reading, we already discussed data and application integration in Chapter 2, introducing the basics and foundations underlying both data and logic mashups. Doan et al. [107] provide a very good summary of the problems and solutions regarding data integration. Alonso et al. [13] surely represent a reference for application integration with a special focus on the case of web services, while Papazoglou [224] more specifically explains the use of web service technologies in practice, e.g., also providing good insight into the problem of correlation in the context of BPEL. Although mashup do not aim at the full power of BPEL, the problem re-presents itself in similar terms

1. Mashup type
2. Component types
3. Runtime location
4. Integration logic
5. Data passing logic
6. Presentation logic
7. Instantiation model

How components and their outputs are visualized in the mashup's UI

Reuse of components’ UIs

Ad-hoc UIs
• **Stateless**
  – No internal state for their execution, ex.: data mashups

• **Short-living**
  – Last the time of a user session, ex.: UI mashups

• **Long-living**
  – Survive across different user sessions, ex.: process mashups
1. Data Mashups

2. UI Mashups
Data Mashups

• **Integrate components at the data layer** of the application stack by fetching data from different data services or Web resources, processing them, and returning an integrated result set.

• **No presentation layer**

• Output: typically **published as a data source**

• Core integration practice: **data integration**
A conceptual model for data mashups

Data mashups fetch data from different sources and integrate them, mix them, filter them, process them, etc., so as to produce a unified data set as output.
Compared with data integration...

- Data mashups are a Web-based form of data integration, intended to solve different problems
- Covering the “long tail” of data integration requirements
  - Very specific reports or ad-hoc data analyses
  - Simple, ad-hoc data integrations providing “situational data” that meet short term needs
  - Non-mission-critical integration requests
Point-to-point data mashups

Basic architecture with direct data passing among components and data processing functions. The mashup control logic establishes the necessary direct point-to-point communications.
Centrally-mediated data mashups

Data are transformed and stored in an integrated data store, and all data processing functions operate on this integrated data store only.
Data mashups with external data processing logic

Besides internal data processing functions, web services or similar are exploited to reuse third-party data processing capabilities and power.
User Interface Mashups

• Component integration at the presentation layer (UI integration)
  – Reusing and possibly synchronizing the UIs of the involved components and mediating possible data mismatches
• Output: a Web application the users can interact with
• Particularly appropriate when components have natively a UI and developing a new UI from scratch is simply too costly
• Mostly client-side, generally short-living
• Different level of complexity: from sharing of a same page layout to complex synchronization/communication patterns
UI mashups without inter-component communication

A specialization of the basic mashup model with new elements: user interface, templates, viewports
The simplest UI mashup: embedding external resources inside own HTML code.
6.5 User Interface Mashups

The simplest UI mashup: embedding external resources inside own HTML code.
Wrapped UI Mashup

Wrappers invokes the original service, interprets and manipulates the retrieved results

- Visualizes data according to a suitable HTML UI
- Captures UI events
- Handles external requests for operations
Wrappers invokes the original service, interprets and manipulates the retrieved results

• Visualizes data according to a suitable HTML UI
• Captures UI events
• Handles external requests for operations
UI mashups with inter-component communication

Three new elements for the synchronization of components: operations, UI events, shared memory
Container-based UI mashups

Assembling a portal page is performed in two phases: (i) generation of markup fragments by portlets and (ii) aggregation of fragments into the portal page. The portlet markup fragments adhere to rules that facilitate content aggregation into portals. Portals interpret the portlet markup code, allocate suitable space for the rendering of each portlet, and generate the composite UI. Portals typically allow users to customize the composite UI (e.g., to rearrange or show/hide individual portlets), and provide facilities like single sign-on and role-based personalization for portlets.

Analogous to Java servlets, portlets implement a standard Java interface (JSR-168 [1]), to enable developers to create portlets that can be plugged into any standard-conform portal. JSR-168 also defines a runtime environment for portlets, the portlet container, and the Java API between the container and the portlets. Figure 6.13 illustrates a typical portal architecture. The portlet container hosts portlets and offers support for their deployment and execution, i.e., it provides the required runtime environment, e.g., with support for persistent storage to store portlet preferences. The portlet container receives requests for the execution of portlets from the portal, where the actual user interaction with the portlet takes place. As such, the portal aggregates the markup of its portlets and manages communications with the portlet container in a centrally mediated fashion. That is, the portlet container is not responsible for aggregating and displaying the fragments produced by the portlets; this is under the responsibility of the portal.

JSR-168 focused on portals that use only portlets installed locally in the portlet container. The Web Services for Remote Portlets specification [264] then standardized the interaction with remote portlets accessed via SOAP.
Different areas of the page correspond to different viewports, each one displaying the content of a widget.
However, the technology is still young and evolving. There are indeed already research works (e.g., [252, 277]) that have started proposing extensions to the widget model to make widgets inter-operable (at least within a same page). Similar discussions are also ongoing in the widget standardization group. The most accredited approach proposes extending the W3C widget model with client-side event generation and handling capabilities. For example, the approach presented in [277] propose the addition of a dedicated Intercom Interface that extends the W3C Widget Interface to support: (i) raising events, i.e., producing messages to communicate internal state changes, (ii) invoking operations on widgets, and (iii) exposing metadata about the events and operations supported by a widget. Event transmission is mediated by a dedicated client-side event bus (see Figure 6.15 or publish-subscribe frameworks (e.g., pmrpc, http://code.google.com/p/pmrpc/) as extensions of the widget runtime environment. Chudnovskyy et al. [76], instead, describe a technique to wrap widgets and to equip them with event handling support, if they don't support events natively.

These extensions enable both the orchestration and choreography of widgets within a same page: In an orchestrated widget integration, it is the central mashup logic that subscribes operations to events, as for example shown in [277]. In a choreographed widget integration, each widget publishes its events based &UI&mashups
Logic Mashups

• Integrate components at the application logic layer, by enabling the **composition of functionality** published by logic or data components, and mediating data compatibility issues if necessary

• Output: a **process** that orchestrates components, in turn published as logic component, e.g., a SOAP web service or JavaScript object

• Covered by traditional practices for Service Composition – no further discussed here
Part III

MASHUP TOOLS AND COMPOSITION PARADIGMS
Learning objective = learn how to obtain the key ingredient for a mashup tool, i.e., the \textbf{mashup language}

1. Mashup design concerns
2. Component abstractions
3. Graphical mashup languages
4. XML mashup languages
5. Other languages
6. Developing languages
7. Reference architecture for mashup tools
MASHUP CONCERNS
Components and component models

UI widget

RESTful Web service

SOAP Web service
Components and component models

Component model 1

Component model 2

Component model 3
Control flow and data flow

1. Configuration and start
2. UI widget
3. RESTful Web service
4. SOAP Web service

Control flow

Data flow

Decisions
Data transformations

- UI widget
- RESTful Web service
- Transformation logic
- SOAP Web service
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 First Line Support team of TIE provides the clients with 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.

Similar to EMML, it is again clear that writing this kind of XML code is not very efficient and that a graphical development tool on top could improve productivity significantly. This is what the OMELETTE Consortium recognized with its implementation of an OMDL-compliant installation of Apache Rave, which is able to store, import and export OMDL mashups. As illustrated in Figure 8.11, given the focus on UI widgets, Rave provides for a "live" modeling environment, in which the user places widgets into the workspace and Rave immediately renders them. That is, instead of having graphical modeling constructs representing UI widgets, Rave directly shows the real widget. This practice is especially effective for users without specific modeling knowledge. What is not visible in the figure is that the extended version of Rave also supports inter-widget communication via a suitable event bus (to be included among the <capabilities> if needed).

OMDL is still in an early stage of its development. However, as a proof of concept the OMELETTE Consortium implemented an import/export filter also for Moodle (https://moodle.org/).
COMPONENT ABSTRACTIONS
Modeling constructs may represent....

**Component instances**
- each component has an own construct

**Component types**
- similar components have the same construct

**Unified component model**
- one construct for all components

---

**Simplicity**

**Domain-specificity**

**Intuitiveness of technicalities**

**Intuitiveness of domain**
Abstracting = wrapping

- **UI wrapper**
  - UI widget
  - JavaScript events
  - Locally installed and running
- **REST wrapper**
  - RESTful Web service
  - HTTP calls
  - JSON formatted data
  - Running remotely
- **SOAP wrapper**
  - SOAP Web service
  - SOAP messages
  - XML payload

Common component model, data format, access protocol
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, i.e., a mashup for business compliance monitoring that leverages on a set of company-internal components. When discussing the mashArt component model (Section 8.3), we have seen that it is based on events and operations. Composing a mashup therefore means connecting events and operations via data flows. The mashup is composed of three UI components (Policy browser, Process browser and Analysis browser and the four Web services (Repository, Engine, Component browser UI component Service component, Data flow connector, UI component

Components in mashArt: apparently two component models
But internally mashArt uses a **unified** component model

The model accommodates: 

- SOAP/RESTful web services
- RSS/Atom feeds
- UI components
GRAPHICAL MASHUP LANGUAGES
Model-driven mashup development

4.3 Metamodeling

The centerpiece of MDSD are the models that are used to design applications in a graphical manner. Metamodeling is the activity that is concerned with the design of the modeling languages that actually enable the abstract development approach that characterizes MDSD. Good modeling languages contain fundamental conceptual, domain, and technological knowledge regarding the development of their target applications and represent the core value of MDSD. Without sensibly and purposefully designed modeling languages, MDSD would not be useful. It is therefore of utmost importance that developers put the necessary effort – and competence – into the design of their modeling languages, especially if we consider that modeling languages typically do not change fast over time and are designed to support the development of multiple applications on top of a same platform infrastructure.

4.3.1 The metalevels

A model, e.g., a UML object diagram, describes the structure and nature of instances, e.g., runtime objects for a given instant of time during the execution of an application. Similarly, a metamodel describes the structure and nature of model elements, i.e., model constructs. The prefix “meta” indicates that we are dealing with models about models. That is, the term is relative, i.e., referring to the model the metamodel is talking about, not absolute.

In Figure 4.4, we show the four metalevels introduced by the OMG, denoted M0, M1, M2, and M3. The metalevel M0 corresponds to concrete runtime instances of an application; the metalevel M1 to the model of the application; the metalevel M2 to the model of the model (the metamodel), i.e., to Meta-metamodel

Fig. 4.4 The four metalevels proposed in OMG’s Meta Object Facility [215].

OMG’s Meta Object Facility (MOF)
Let’s proceed by example...

Let’s design a simple data mashup language...

1. Integrate RSS feeds
2. A Union operator merges feeds
3. A Filter operator filters items by conditions
4. A Sink component ends processing
5. Data flow connectors propagate data
8.4.1 A simple example

Both to recall the basic metamodeling 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.
Model (abstract syntax) = instance of metamodel

NY Times: Name

http://rss.nytimes.com/...

RSS feed

DF1: Data flow connector

source

target

U1: Union

source

DF3: Data flow connector

target

BBC News: Name

http://feeds.bbci.co.uk/news/rss.xml

RSS feed

DF2: Data flow connector

source

target

F1: Filter

source

DF4: Data flow connector

target

S1: Sink

title contains 'IT':

Filter condition

In line with the MDSD approach described in Chapter 4, we can now represent a model instance via an abstract syntax, such as a UML Object Diagram. Figure 8.4 proposes a possible mashup model in abstract syntax: We have two RSS feeds (NY Times and BBC News), which we merge into one single feed using a union operator, followed by a filter operator, which lets pass only items that contain the substring “IT” (for simplicity, we express conditions in natural language). Finally, a sink component indicates the end of the processing logic and makes the result of the mashup available as RSS feed accessible via the Web.

Figure 8.4 is a full-fledged model of a mashup that complies with our initial requirement, implemented in the metamodel of Figure 8.3. However, the reader will easily agree that the abstract syntax of the model does not help the readability and understandability of the model. The convention of using a UML Object Diagram may be good as first check of the correctness of a metamodel and its corresponding modeling language, or to understand how to implement code generators or model interpreters, but it certainly is not meant to be used to really model mashups. For this purpose, in Figure 8.5 we invented a graphical, concrete syntax, whose aim is to make the semantics of modeling constructs intuitively understandable: Data flows from Model & (abstract & syntax) & = instance & of & metamodel &
Model (concrete syntax) >> Human readable

Same model as before!
Let’s **reverse-engineer** Yahoo! Pipes
And here a possible metamodel

A pipe must contain exactly one pipe output component, which cannot be the source of a data flow connector.
XML MASHUP LANGUAGES
EMML, the Enterprise Mashup Markup Language

EMML, the Enterprise Mashup Markup Language [http://mdc.jackbe.com/prestodocs/v3.7/emml/mashup-library-intro.html]

```xml
<?xml version="1.0"?>
<mashup name="Mashup News"
   xmlns="www.openemml.org/2009-04-15/EMMLSchema"
   ../../../schema/EMMLSpec.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <variables>
     <variable name="news" type="document"/>
   </variables>
   <output name="result" type="document"/>
   <directinvoke endpoint="http://rss.nytimes.com/services/xml/rss/nyt/Technology.xml" method="GET"
     outputvariable="news"/>
   <filter inputvariable="news"
     filterexpr="matches($news/rss/channel/item/description,""Mashup")"
     outputvariable="result"/>
 </mashup>
```

Variables

Mashup output declaration

Feed invocation

Filter over items

Connection with output channel
**Software AG Presto** implements EMML

JackBe has recently been acquired by Software AG, and many of the former online resources are no longer accessible
OMDL: the Open Mashup Description Language

```xml
<?xml version="1.0"?>
<workspace xmlns="http://omdl.org/"
    <goal>Illustrate a simple OMDL example</goal>
    <identifier>http://repo.omdl.org/mashups/...</identifier>
    <title>An OMDL example</title>
    <description>...</description>
    <creator>Florian Daniel</creator>
    <date>2013-10-08T14:23+37:00</date>

    <app id="http://repo.omdl.org/mashups/alice/CallFromMap/1">
        <type>MAP</type>
        <link rel="source" href="http://repo.omdl.org/apps/map/MyFancyMap" type="application/widget"/>
        <position>TOPLEFT</position>
    </app>
    ...
    <app id="http://repo.omdl.org/mashups/alice/CallFromMap/2">
        ...
    </app>

    <capabilities>
        <gps mandatory="true" accuracy="100"/>
    </capabilities>

    <layout>GRID</layout>
    <stylesheet>
        http://repo.omdl.org/mashups/alice/3/_data/special.css
    </stylesheet>
```

Usual metadata

App/widget declarations

Type

URL

Position

Capabilities required by mashup

Layout template

UI mashups
The **OMELETTE Apache Rave** environment has OMDL-compliance workspaces.
DEVELOPING MASHUP LANGUAGES
Conceptual development
of mashup languages/platforms  [Soi2014]

Observation: All mashup languages share similar features

Idea:
1. Extract/isolate features
2. Express features as reusable mashup language patterns (XSD)
3. Implement a library of features (XSD)
4. Identify conflicts and inclusions (simple rules)
5. Develop a runtime environment that supports all features
6. Develop new languages by assembling features
   - Mashup language
   - Component description language
7. Customize the runtime environment with new language
8.6 Developing Mashup Languages

Conceptual Design of Sound, Custom Composition Languages

Custom composition language

Composition feature

Implemented as

Feature constraint

Has

0..N

1..N

Constrains

Implemented as

Component feature

Control flow feature

Data passing feature

Presentation feature

Collaboration feature

Feature reference specification

Derives from

0..1

1..N

Integrates

0..1

1..N

Generic composition language meta-model

Integrates

0..1

1..N

Generic component descriptor meta-model

Supports

1..N

Supports

1..N

Based on

Based on

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.
Figure 7. The customizable mashup development environment. The two screen shots illustrate how the environment changes in function of the chosen mashup language.

(a) An instantiation of the mashup editor based on the data flow paradigm.

(b) An instantiation of the editor using a control flow paradigm with global variables for data passing.
Fig. 8.13 The generic mashup language model bringing together the most common mashup features [253].

Development: unlike, for example, Web services, language patterns are not independent of each other. That is, the reference specifications of different composition features may overlap (e.g., interacting with a SOAP service is very similar to interacting with a RESTful service), include other features (e.g., the data flow paradigm generally subsumes the presence of data source components), or exclude others (e.g., the data flow paradigm does not make use of variables). This asks for a thorough design of the language patterns and their mutual interaction points, a task that the authors solve by mapping each composition feature into the generic mashup language model (see Figure 8.13), which (i) integrates all basic language constructs syntactically, (ii) allows the definition of composition features as language fragments on top, and (iii) guarantees that fragments are compatible by design.

It is important to note that the model in Figure 8.13 and the language generation platform are extensible, that is, the proposed approach allows for the addition of new constructs and features, if these extend the model without altering the logic of the existing constructs, if each new construct is accompanied with a reference implementation for the runtime environment, and if the features are equipped with suitable reference patterns. The model aims to cover as many features as possible, but it is of course not feasible
Example of feature specification

```xml
<feature name="data_flow" label="Data flow">
  
  <description> The composition paradigm is data flow, that is, it is possible to explicitly define the flow of the data among components operations. In this case the data passing and the control flow overlap since operations triggering depends on the data flow. </description>

  <specification>
    
    <include fragments="dfConnectorDef, dfConnectorType, dfSourceOutputParameter, dfTargetInputParameter" />

  </specification>

  <constraints>NOT(control_flow)</constraints>

</feature>
```
Runtime environment (operational semantics of language)

Domain expert

Mashup user

W3C widget
JS library
mashArt UI component

UI events

Client-side runtime environment

W3C widget adapter
JS adapter

Client-side engine

initiates UI mashups

C/S communications

Web service interface

DMT configuration package

Mashups

Server-side runtime environment

Server-side engine

REST adapter
RSS adapter
Atom adapter
SOAP adapter

The Web

Components

Application or web service

uses

uses

uses

uses
8.6 Developing Mashup Languages

Conceptual development example: Yahoo! Pipes

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. request_response
9. max_1_operation_per_component
10. min_1_output_param_per_operation
11. max_1_output_param_per_operation
12. min_1_input_param_per_operation
13. max_N_input_param_per_operation
14. manual_input
15. configuration_param
16. branch

For instance, if we wanted to compose a mashup language that is able to equip each component with authentication credentials so that the runtime environment can automatically authenticate with the components on behalf of the user (e.g., to implement a single sign-on feature), this would not be possible without suitable extensions. So far, the approach has been tailored to and tested with open APIs and services that don’t need this kind of authentication.

8.6.2 Usage example

As an example of how the conceptual development of mashup languages may happen in practice, in the following we derive part of the mashup language underlying Yahoo! Pipes from an example modeled in its graphical editor.

Pipes is based on the data flow paradigm. It supports data component and service component types to retrieve and process data, respectively. Specifically, data sources may be rss for data or atom for data, while the only supported service component type is REST for service. Each component in Pipes provides exactly one function, that is, each component represents one single operation: max 1 operation per component. All operations are of type request-response (request response for data and request response for service). Each operation may have one or more inputs (max N input param per operation) but one and only one output (max 1 output param per operation). Manual inputs are used to fill the values of input fields (manual input), i.e., of configuration params.
Attention: this model is by design different from the metamodel presented earlier!
OTHER MASHUP LANGUAGES
data views have been constructed, they need to be displayed on the
ically construct data views over a given ER-based schema. Once
3.2 Supporting complex data in spreadsheets
ately showed to the users (hence the name
or descending order. The result of all these operation is immedi-
instances in a data view according to their attributes in ascending
given filter predicate. The maximum number of instances displayed
instances of an entity to be retrieved to the subset that matches a
by ticking corresponding check boxes, (ii)
Figure 3: Spreadsheet-based mashup for the reference scenario
contains only news related to stock data. Specifically, this user in-
attributes and related entities of
as an expandable child node which, when expanded, presents the
Stock
Figure 3

Spreadsheet-based mashups [Kongdenfha2009]
Spreadsheet-based mashups [Kongdenfha2009] Social Spreadsheet [Jara2013]
NaturalMash [Aghaee2013] = controlled natural language
The WYSIWYG composition editor of the PEUDOM platform enables non-programmers to create UI mashups easily. The platform further assists users in defining component operations and event-driven logic to expose services and make them observable through the component UI. Proper wrappers add to the synchronisation of component UIs.

Each component is modeled as a provider of events and operations, produced by events raised during user interaction with the components. Component operations can be modified by interacting with the component or by defining sensible event bindings based on data compatibility. The PEUDOM platform also enables the synchronization of components at the UI level.

The user can enrich the default synchronization behaviour, and define further component couplings by selecting possible behaviours that the two components have to show within the final application.

PEUDOM [Matera2013] = live, visual programming
8.7.1.1 Design

The previous sections reviewed the core concepts driving the development of mashup editors. We specifically focused on mashup metamodels and mashup languages, as these both express a wealth of mashup knowledge and also have a high didactic value. We did not focus much on the user interface paradigms that can be used by mashup editors to communicate their internal metamodel/language to the user of the editor. Not all editors directly expose their plain language or an abstract modeling notation to their users. Most editors adopt proprietary, domain-specific notations that aim to improve the intuitiveness of the editor.

For instance: Swashup\(^{186}\) leverages on traditional, manual coding of mashups; domain-specific code is then translated into Ruby on Rails. Mashlight\(^{26}\) and mashArt\(^{90}\) propose an own graphical modeling notation for Web services, RSS/Atom feeds and UI widgets. Kongdenfha et al.\(^{168}\) propose the use of spreadsheets for the design of mashups, where functions inside

Fig. 8.15 Conceptual reference architecture of a mashup platform articulated into front-end, back-end and persistent data store.
Part IV

MASHUP QUALITY
Learning objectives

1. Component quality
   – Definition of the main data quality dimensions to evaluate mashup components

2. Composition issues
   – Issues related to the assessment of the quality of composed application

3. Mashup quality
   – Definition of the data quality dimensions to evaluate mashup applications
The importance of quality

Garbage in → garbage out
“Even though quality cannot be defined, you know what it is....”

Robert Pirsig
Is Quality Measurable?
Quality Assessment

What?
Dimensions

How?
Metrics

We need a quality model!
Mashups

“Mashups are Web applications that integrate inside one web page *two or more* heterogeneous resources....”
Integration of two or more heterogeneous sources...
Quality of a composed object

The quality of the composed objects depends only on the quality of the components?
FROM THE COMPONENTS ...A QUALITY MODEL
The structure of an API
Quality contributions

- Software quality dimension: ISO standard
- Contributions addressing quality of software components: complexity, modularization, cohesion, coupling
The quality model an overview...

Component quality

API quality
- Functionality
  - Interoperability
  - Compliance
  - Security
- Reliability
- API Usability
  - Maturity
  - Reputation
  - Learnability
  - Operability

Data quality
- Accuracy
- Timeliness
- Completeness
- Availability

Presentation quality
- Presentation Usability
- Accessibility

[Cappiello2009]
API quality - Functionality

Interoperability

Protocols
Language
Data formats

Compliance
API quality – Functionality

Security

Security requirement = No auth no SSL
1 1 1 1 1
Security requirement = Developer key over SSL
1 2 3 4 4

No SSL support
With SSL support

Authentication model

<table>
<thead>
<tr>
<th>Authentication model</th>
<th>No authentication</th>
<th>API key</th>
<th>Developer key</th>
<th>User account</th>
</tr>
</thead>
<tbody>
<tr>
<td>No SSL support</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>With SSL support</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
API Quality - reliability

Maturity\textsubscript{comp} = \max(1 - \frac{\text{CurrentDate}_{\text{comp}} - \text{LastUseDate}_{\text{comp}}}{\text{CurrentDate}_{\text{comp}} - \text{CreationDate}_{\text{comp}}} ; 0)\\

Age \quad Usage \quad Maintainance

We will discuss presentation usability later in this section.
API usability - operability

(a) A metric to measure operability of API types

(b) A metric to measure the operability of data formats
API usability - learnability

Examples

```
import java.util.Scanner;
import org.scribe.builder.*;
import org.scribe.builder.api.*;
import org.scribe.model.*;
import org.scribe.oauth.*;

public class TwitterExample{
  public static void main(String[] args){
    OAuthService service = new ServiceBuilder()
      .provider(TwitterApi.class)
      .apiKey("01cbcaNg2z67r8fUAlM5Q+j")
      .apiSecret("SCCAdU6C6LXlaZxH3iDwDfpmUJVlyj4eM2ZlXfLlJv396")
      .build();
    Scanner in = new Scanner(System.in);
    Token requestToken = service.getRequestToken();
    System.out.println(service.getAuthorizationUrl(requestToken));
    System.out.println("And paste the verifier here: ");
    Verifier verifier = new Verifier(In.nextLine());
    Token accessToken = service.getAccessToken(requestToken, verifier);
  }
}  ```
Data Quality

- Accuracy
- Completeness
- Availability
- Timeliness
Presentation Quality

Usability

Accessibility

Reputation
TO THE COMPOSITION...A QUALITY MODEL
Mashup quality: main aspects

Mashup component: Google Maps

Mashup component: Flickr

Garbage in \(\rightarrow\) Garbage out

Garbage inside \(\rightarrow\) Garbage out

Mashup application:
e.g., BrusselStripstad.be
Quality assessment?

Mashup component: Google Maps

+ 

Mashup component: Flickr

= 

Mashup application: e.g., BrusselStripstad.be
Quality assessment: a first experiment

Mashups are accessible as normal Web pages... ...can we use the same models and tools developed for quality assessment of traditional web pages?

• We assess (by using automatic tools) the quality of 68 mashups on the basis of four criteria:
  • Usability: measures the ease of use of the mashup. (SiteAnalyzer)
  • Readability: measures how easy or difficult it is to read and understand the text rendered in the mashup. (Juicy Studio)
  • Accessibility: measures how well the mashup complies with the W3C web accessibility guidelines. (SiteAnalyzer)
  • Performance: measures the loading time of the mashup till all elements of the application are rendered in the page. (Pingdom)
Results of the experiment: five "best" and five "worst" mashups.

- Are these results reliable? We conducted five independent evaluations by manually inspecting the same mashups and we compared the two evaluations and we found…
... a counterexample...
Mashup quality: the need for a quality model

• High focus on composition aspect in mashups:
  • Data integration
  • Service orchestration and UI synchronization
  • Layout

• The success of a mashup is certainly influenced by the added value that the final combination of components is able to provide.
Composition patterns [Cappiello2010]

http://dailymashup.com/

http://www.housingmaps.com/

http://immo.search.ch/

(a) Slave-Slave pattern

(b) Master-Slave pattern

(c) Master-Master pattern
The quality model

[Cappiello2011]
Data Quality

Incompleteness

Inaccuracy
Presentation quality

- **Usability**
  - **traditional** dimensions such as orientation, users control, predictability, layout consistency
  - **Learnability**: the mashup features should be visible enough and the corresponding commands should be self-expressive so that even naive users can easily master the mashup execution.
  - **Layout consistency**

- **Accessibility**
  - Accessibility criteria do not need to be specialized for mashups.

- **Usability**
  - navigability and richness of links, or any other criteria addressing the hypertext structure
  - readability, cohesion or coherence
Composition Quality: added value

- The added value of the composition can be related to the amount of provided **features and/or offered data**. The mashup has an added value if it provides at least more functionality or data than the ones provided by its components.
Composition Quality: other dimensions

- **Component suitability**: it refers to the appropriateness of the component features and data with respect to the goal that the mashup is supposed to support.

- **Component usage**: it may happen that, even though a component is very rich from the point of view of data and functionality, it is improperly used within a composition.

- **Consistency**: poor quality compositions can also be caused by inconsistencies at the orchestration level.

- **Availability**: the degree in which the mashup can be properly accessed during a given time interval. It depends on the availability of the components and on their role in the composition.
Remember....

*Garbage in ➔ garbage out*
OUTLOOK
Growth

Threats

Directions
Growth

- Availability of components
- Benefit of reuse
- New technologies
- End-user skills
- End-user innovation

Directions

- Threats
Growth & Threats & Directions

Fast evolution
High complexity
Development integration
Nunfunctional properties
Growth

Threats

Directions

Standardization

Intuitive technologies

Quality assurance

Assistance
Main reference


Other references


