Web Mashups

Integration the Web 2.0 way

Florian Daniel (daniel@disi.unitn.it) – April 30, 2009
Last time

• Definition of mashups
• Technologies
• Manual development
• Partially assisted development

Today
• Assisted/aided development
• Own mashup research at UNITN
Fully assisted development

- Mashup tools/platforms
  - **Simplify** the overall development process
  - Provide easy-to-use development instruments
  - Provide dedicated execution environments
  - Support the whole lifecycle of mashup applications
  - Enable even the less experienced user to mash up own applications

- Let’s see some representative examples
  - Yahoo Pipes, Intel Mash Maker, Microsoft Popfly, JackBe Presto (yet, there are many others)
• Powerful, hosted **data mashup** tool for the processing of
  • RSS/Atom feeds
  • XML/JSON data resources/services
• Targets skilled users and **programmers**
• **Data flow** approach (pipes)
• No support for **user interface** design
Client-side **browser extension** for interactive mashup development
- Data extracted from annotated web pages
- Widgets (UI components) for data visualization
- Copy/paste of Web contents into other Web pages
- Targets **average Web users** and programmers
- Data passing through **environment variables**
- No support for **service components**
• Highly interactive, hosted mashup platform for consumer mashups
  • Mashup “blocks” for data, application logic, and UIs
  • Mainly JavaScript blocks
  • Comes with own block builder
• Targets advanced Web users and programmers
• Data passing by coupling components and mapping outputs to inputs
• Still weak support for UI components
• Full-fledged enterprise mashup platform with desktop integration
  • Main focus on data mashups
  • Support for web services and (local) spreadsheet files
  • Separate layout support for UIs (mashlets and portals)
• Targets advanced users and programmers
• Data flow logic
• Still limited layout capabilities
So, now it’s your turn!

• Let’s do some **exercises** with Dapper and Pipes...

• **Dapper**: try to extract authors, title and reference of the publications on my web site (floriandaniel.it)

• **Pipes**: try to merge two different RSS feeds from Springer and from ACM (regarding scientific publications)
Our own research on mashups

- **UI integration**
  - Stand-alone web apps as **UI components**
  - **Synchronization** among components

- **Universal integration**
  - UI, application logic, and data components
  - One **component model**: abstract components, highlight similarities
  - One **composition model**: one formalism for synchronization and orchestration
  - **Hosted** development and execution
UI integration: visual editor

- List of application components available for the mashup. Additional components may easily be loaded into the editor by referencing the respective online resource.
- Mahup logic modeling canvas.
- Graphical model of the composition logic.
- Tabs that allow the designer to switch between different views (e.g., composition logic vs. layout) on the composite application under development.
- Deployment
- The mashup application running in a standard web browser

The mashup application running in a standard web browser
Universal integration: goals and challenges

- Creating a universal **component model**
  - Abstract **common features** of UI and services
  - Recognize **differences** wrt traditional service integration
- Identifying the key abstractions for a universal **composition model** that is simple but effective
  - Data flows
  - Stateful vs. stateless components
  - Components with and without UI
- Offering **integration as a service**
  - **Hosted** development, execution, and analysis
Guiding principles

• **Universality**: UI, application logic, and data
• **Synchronization and orchestration**: bring together UI and application/service logic
• User and programmatic **input**: UIs + APIs
• **Lightweight** composition model: 2 pages spec, no transactions, no exceptions/compensations
• Complexity **inside** components
• **Extensibility** and openness
• **Standard** layout and graphics tools
Universal integration

- Service component
- Data flow connector
- UI component
- Component browser
- Composition canvas
The mashArt component model

- **State**
  - UI components: given by the “state” of the user interface
  - Services: internal “program state” (e.g., of an order service)
- **Events**
  - Publish state changes (data formatted as name-value pairs)
- **Operations**
  - Change state or perform operation invocations
- **Configuration properties**
  - E.g., layout settings or authentication options
  - Constructor parameters
The mashArt component model

The mashArt component model consists of several components and relationships. Here are the details:

- **State Variable**
  - Name
  - Value
  - Type: 0..N

- **mashArt component**
  - Name
  - Binding URL
  - Type: 0..N

- **User interface**
  - Output

- **Event**
  - Name
  - Type: 0..N

- **Parameter**
  - Name
  - Type
  - Value
  - Type: 0..N

- **Constructor**
  - Mandatory input
  - Optional input
  - Constant input

- **Operation**
  - Name
  - Reference
  - Type: 0..N

- **Request-response**

- **One-way**

The diagram illustrates the relationships and types of components involved in the mashArt model.
Modeling mashArt components

- **Common** features of components
  - Data flows + events + operations
- **Distinguishing** features among components
  - Stateful vs. stateless + UI vs. service
- Graphical modeling **notation**

[Diagram showing data flows and operations for different types of components]
Describing mashArt components

- mashArt Description Language (MDL)

```xml
<?xml version="1.0" encoding="utf-8" ?>
<mdl version="0.1">
  <component name="Policy" binding="component/UI" stateful="yes" url="http://mashart.org/registry/234/policyBrowser.js">
    <types>…</types>
    <event name="PolicySelected">
      <output name="policy" type="xsd:string"/>
    </event>
    <operation name="ShowPolicy" ref="showPolicy">
      <input name="policy" type="xsd:string"/>
    </operation>
    <constructor>
      <input name="NumVisible" type="xsd:integer">5</input>
      <input name="StartPolicy" type="xsd:string" optional="yes"/>
    </constructor>
  </component>
</mdl>
```
Universal composition model

- **Event**-based + **flow**-based composition
  - Conciliates synchronization and orchestration
  - Linking of events to operations (“basic listeners”)
  - Workflow fragments (“complex listeners”)
- Flow control via **conditional** execution of operations
  - Allows specification of split/join constructs
- As **simple** as possible, yet as **complete** as possible
  - No complex modeling constructs
  - Complex behaviors/features can be plugged-in via components (>> complexity inside components)
Universal composition model
Modeling universal compositions

- Example of a universal composition model:
Universal composition language (UCL)

• Component and type declarations:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ucl version="0.1">
  <composition name="BCM application">
    <components>
      <component id="Policy" mdl="http://mashart.org/registry/...">
        <input name="NumVisible"> 10 </input>
      </component>
      <component id="Process" mdl="http://mashart.org/registry/..."> ...
      </component>
      ... //declaration of Analysis, Repository, Engine, Analyzer and Mail omitted
    </components>
    <types>... </types>
  </composition>
</ucl>
```
Universal composition language (UCL)

- Basic listeners:

```xml
<listener id="l2">
  <event id="e2" component="Process" name="ProcessSelected"/>
  <operation id="o6" component="Analysis" name="ShowAnalysis"/>
</operation>
</listener>

<listener id="l4">
  <event id="e4" component="Analysis" name="ViolationDetected"/>
  <operation id="o9" component="Mail" name="SendMail"/>
</operation>
</listener>
```
Universal composition language (UCL)

• Sequencing, branching, joining:

```xml
<listener id="l1">
  <event id="e1" component="Policy" name="PolicySelected"/>
  <operation id="o1" component="Repository" name="GetProcsByPolicy">
    <input name="pol" src_id="e1" src_name="policy"></input>
  </operation>
  <operation id="o2" component="Engine" name="GetProcs">
    <input name="procs" src_id="o1" src_name="proc"></input>
  </operation>
  <operation id="o3" component="Process" name="ShowProcesses">
    <input name="process" src_id="o2" src_name="procs"></input>
  </operation>
  <operation id="o4" component="Analyzer" name="GetResults">
    <input name="processes" src_id="o2" src_name="procs"></input>
  </operation>
  <operation id="o5" component="Analysis" name="ShowAnalysis">
    <input name="results" src_id="o4" src_name="analyses"></input>
  </operation>
</listener>
```
Hosted execution environment
Hosted execution environment

• Development challenges:
  • Seamless integration of **stateful** and **stateless** components and of **UI** and **service** components
  • **Short-living** and **long-running** process logics in the same environment
  • **Distribution** of execution taks over client and server
  • Transparent handling of multiple **communication protocols**
Analyzing mashup tools

Determines the **nature** of components and influences how components can be glued together.

Determines how components are **integrated** to form the mashup, assuming components are readily available.

**Component model**

**Composition model**

**Development environment**

**Runtime environment**

**Assists** the developer in the mashup process and eases development.

Enables the **execution** of mashups and determines how mashups are delivered to their users.
Component model

- **Type**
  - Data (DA) vs. application logic (AL) vs. user interface (UI)

- **Location**
  - Local vs. remote

- **Direction of interaction**
  - One-way vs. two-way

- **State**
  - Stateful vs. stateless

- **Behavior**
  - Active vs. reactive
Composition model

- **Type**
  - Data (DA) vs. application logic (AL) vs. user interface (UI)

- **Orchestration style**
  - Flow-based vs. event-based vs. layout-based

- **Data passing style**
  - Data flow vs. blackboard (without vs. with shared memory)

- **State**
  - Stateful vs. stateless

- **Instance model**
  - Instance-based or continuous
Development environment

- **Target users**
  - Web users vs. tech-savvy users vs. programmers

- **Interface paradigm**
  - Visual drag-and-drop vs. textual editors vs. combinations

- **Type of support**
  - Composition only vs. composition + components vs. component only

- **System requirements**
  - Hosted, web-based vs. standalone
  - Additional modules, plug-ins, or browser features
Runtime environment

• **Deployment model**
  • Complied (web app based) vs. interpreted (engine-based)

• **Execution location**
  • Local vs. remote vs. hybrid

• **System requirements**
  • Browser plug-ins or extensions?

• **Scalability**
  • Number of data sources, in the number of models (compositions), or in the number of users
A new development paradigm?

• Characteristics of modern web applications
  • Fast development cycles (Internet time)
  • Incremental development (prototype-based)
  • Continuous online evolution

• The software life cycle of modern web applications is no longer captured by traditional life cycle models (e.g., the spiral or the waterfall model)

• And what about user-driven composition of web applications and mashups?
The evolution life cycle model

• A model for modern web applications (e.g. Google):
The mashup life cycle model

- A model for the **mashups to come:**

  - **Mashup idea**
  - **Discovery and selection**
  - **Mashup**
  - **Evolution**
  - **Deployment**
  - **Usage and maintenance**
  - **Dismissal**
  - **Online mashup application**

- End users doing “development”: this indeed will be a **paradigm shift** in web development!
Applicability of mashups

• But what about the utility of mashup applications?
  • Mashups are still mostly 1-page apps...
• Only very few innovations are really breakthroughs, most innovations only create little value
• Perfectly understanding customer needs, in order to customize software and satisfy as much users as possible, is costly – if not impossible
• Mashups may leverage “user innovation”:
  • Users themselves know best what they want
  • Mashups enable them to build their own applications
The long tail of the SW market

Using traditional approaches, just the most important projects can be implemented.

With Mashups, the long tail of projects can be implemented.
Important concerns not yet addressed

• **Security**
  • AJAX poses novel challenges, authentication (OpenID, ...)

• **Compatibility**
  • Not only browsers, but also mashup platforms

• **Performance**
  • Hard to guarantee constant performance levels on the client

• **Quality**
  • We’ve analyzed data and APIs on programmableweb...

• **Attribution/Legality**
  • Terms of use, owners of contents, licenses,...
Conclusion

• Mashup tools will significantly **ease** the development of mashup applications
  • Easy-to-use **development** environments (even to end users!)
  • Libraries of ready **components**
  • Easy **deployment** and **execution**

• However, they are still in an early stage
  • Still in alpha/beta releases (bugs!)
  • Not yet ready for real business applications
  • Not yet ready for the unskilled end user

→ **There is still a lot of work and research to do!**
Wanna work on mashups?

• If you are interested in doing research in this area, feel free to contact us. We are working on:
  • Models and languages for mashups (universal integration)
  • Development of components and adapters
  • Ajax development environment
  • Client-side/server-side execution environments
  • Reporting and analysis tools for end users
  • ... (your ideas count as well)

→ daniel@disi.unitn.it, casati@disi.unitn.it