Lively Fabrik - A Web-based End-user Programming Environment

Jens Lincke[1] Robert Krahn[1] Dan Ingalls[2] Robert Hirschfeld[1]

[1] Hasso-Plattner-Institut, University of Potsdam
{*jens.lincke, robert.krahn, hirschfeld*}*@hpi.uni-potsdam.de*

[2] Sun Microsystems Laboratories, Menlo Park
dan.ingalls@sun.com

Abstract

Lively Fabrik is a Web-based general-purpose end-user programming environment. Based on the Lively Kernel, Lively Fabrik extends the ideas of the original Fabrik system by empowering end-users to create interactive Web content entirely within their Web browsers. Web applications created with Lively Fabrik typically combine Web sources, data manipulation, and interactive user interface elements. The result can be a Mashups, but due to the powerful underlaying system, any general-purpose application. Connecting components with wires and scripting components is all that is needed to do so.

1 Introduction

More and more end-users treat the Web browser as their operating system [17]. Besides reading the Web, they manage their personal correspondence via email, contribute to Wikipedia articles, or collaborate in online spreadsheets. The creation of Mashups [12] by combining Web-based services from different sources into a single application dedicated to a particular task, and the customization of existing Web applications like content management systems are other examples of end-user development that have become popular.

Even though development support in these areas exists for restricted domains, usability and simplicity offered to the end-users are still lacking. One disadvantage of Web-based applications compared to their non-Web counterparts is that the Web programming model is still deprived of the rich capabilities that come with traditional desktop environments.

In an effort to remedy this situation, we have designed and implemented Lively Fabrik, a rich Web-based end-user programming environment built on Lively Kernel [5] and based on Fabrik [6]. Lively Fabrik is an environment that empowers end-users to interactively create their own dynamic Web applications from within their Web browser instantly and without the need to upload or download anything.

We want the user to create a user interface, to program, and to play with the final product in one environment and thus make development iterations simple and short. Because we want to make the application creation as direct as possible, we do not separate the programming process from the process of creating the user interface as it is good practice in software engineering.

Graphical and textual scripting languages are widely used in end-user programming scenarios. Data-flow paradigms have their merits when it comes to dealing with large amounts of data and when they fit naturally into domains where data is retrieved, filtered and combined. We combine these two approaches in Lively Fabrik: we help end-users structuring their programs with the data-flow paradigm and provide scripting for cases where data-flow is complicated or limiting.

Lively Fabrik is Web-based, because many end-users are not allowed to install third party software on the computer they are using but have a modern Web browser at their hands. This may be because they are in an Internet café or they are using a pool computer. Other positive aspects of the Web-based approach is the capability for automatic software updates as well as collaboration between the end-users [15] and sharing of content.

The remainder of the paper is organized as follows. Section 2 gives a short introduction into end-user Web application development. Section 3 describes the visual Web-based end-user programming environment Lively Fabrik. Section 4 discusses implementation details. Section 5 shows the usage of the environment by creating a weather widget Mashup. Section 6 discusses related work and Section 7

makes a summary and gives an outlook.

2 End-user Web Application Development

The state of the art tools for authoring content in the browser include native Web applications like Wikis and content management systems and Web versions of typical desktop applications like word processors, spreadsheets, or drawing programs. These Web applications allow the often collaborative creation of passive media like texts, tables, and pictures. Other tools are specialized on online creation of Web pages and special applications like Web shops. All these online applications are limited when it comes to programming in the browser and specifically programming for end-users in the browser.

Web-based programming is different from programming on personal computers. Programmers have to cope with client/server aspects, security limitations, less CPU cycles to burn, and inferior libraries. For example, the access to local files and services on the client side is restricted because of security reasons. Some of these issues are addressed by advancements in browser technologies like faster JavaScript virtual machines and more powerful browser APIs. Other issues such as the security limitations have to be dealt individually. For example, if JavaScript applications need access to the Internet a solution is to provide a Web proxy on the server of the original JavaScript source file.

Building real applications for a Web browser in itself was a challenging activity, but it gets easier with the evolution on browser technology. We want to go further and enable end-users to create their own applications, without having to struggle with the limitations of the Web environment.

An important domain of end-user programming in the browser is the creation of Mashups. A Mashup (for an overview see [12]) is a recombination of different Web pages or services in a new and often unanticipated way. It typically consist of a script running on a server, that fetches data from different sources and produces new Web content.

We want to create an end-user development environment to create simple Web-applications like Mashups or widgets.

3 Lively Fabrik - Visual Web-based End-user Programming

Lively Fabrik brings the ideas of Fabrik [6, 9] and Lively Kernel together to create an environment for end-users, where they can build their Web applications in a very direct and responsive manner.

We combine scripting and data-flow programming as well as a rich graphical environment to provide a Web-based end-user programming environment for dynamic Web pages.

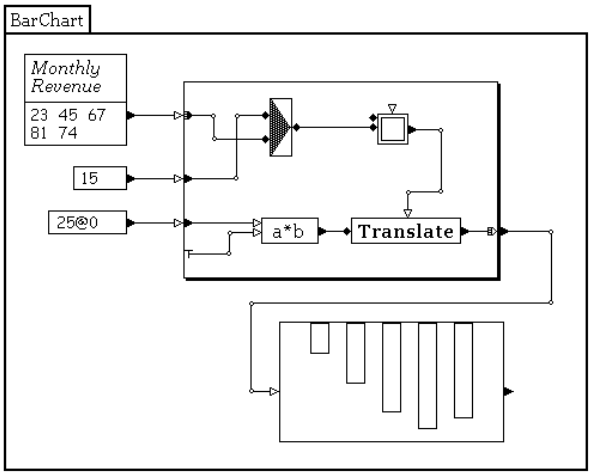

Figure 1. A visual program in the original Fabrik that generates a bar chart (Source [6])

3.1 An Environment for End-user Programming

Lively Fabrik is designed as an uncomplicated environment for creating Web Applications. User interface elements and program behavior are authored in one place.

Furthermore, we want to reduce the complexity by providing a distinct and small set of concepts and principles to keep things simple. Our building blocks are connected components where data flows from one component to the other through wires.

3.2 Components

Applications in Lively Fabrik can be build from very few elements, thus lowering the conceptual complexity of programming. The main elements are components, pins and connections between them. Components manipulate data, create data or trigger side effects. Pins define the interface to components and are the endpoints of connections. Lively Fabrik defines a set of primitive components which are not created in Fabrik itself. These components are:

Input and Output *TextComponents* display strings which can be both user input or data from other components.

ListComponents have an input pin 'List' and an output pin 'Selection'. They show arrays and allow the user to select elements from it.

ImageComponents load and display images from specified urls.

Scripting *FunctionComponents* allow the evaluation of JavaScript. The scripts can access any number of pins for in

and output. By default there is a pin 'Input', which is a parameter to the function, and a pin 'Result', which automatically gets the return value of the JavaScript code assigned.

Wrapping *PluggableComponents* are used to wrap normal Lively Kernel widgets that are dropped into a Fabrik. Widgets have a formal interface from which a pins can be generated and enable the widget to act as a component in Fabrik.

Web Requests *WebRequestComponents* retrieve data from URLs. They could be implemented with *FunctionComponents* but are provided for convenience reasons.

Encapsulation With *FabrikComponents* users are empowered to create their own components. *FabrikComponents*act as containers for other components and encapsulate them: Components inside a *FabrikComponent* cannot directly be connected to outside components. The only means to do this is a indirect connection via pins of the *FabrikComponent*. *FabrikComponents* can be collapsed for hiding internal components. When a user frame is added to a *FabrikComponent*, components in this user frame are not hidden.

3.3 Pins and Connectors

When the users want to connect two components, they click on the pin of the first component and then on a pin of the second component to establish a directed connection between those components (respectively their pins). A connection between pins means: when the value in the pin of the first component changes, it is copied to the other pin. Thus the connection is directed. Bidirectional connections are created by reverse connecting two pins. Connections are not allowed to cross the borders of a *FabrikComponent*, this maintains order and avoids 'wire-chaos'.

3.4 Enhancing the Data-flow Principle with Scripting

We have chosen wiring components as visual data-flow paradigm because it is intuitive and most users think in terms of data going from one place to another [2, 7].

Lively Fabrik does not restrict the type of data that flows from pin to pin. A component can write references into pins, that makes it possible to work with complex objects like Morphs and does not restrict the graphical languages to primitive data values like numbers, strings, points or rectangles.

Although the data-flow principle is very expressive, data-flow programming languages have their shortcomings.

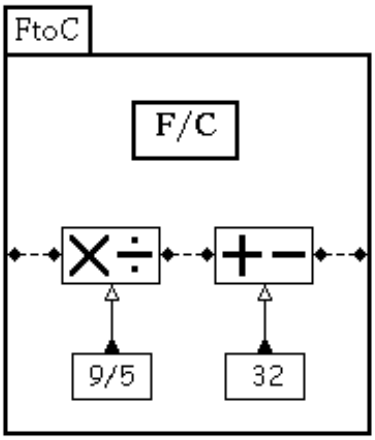

Figure 3. Bidirectional Fahrenheit Celsius converter in the original Fabrik (Source [6])

The original Fabrik demonstrated bidirectional data-flow with a Fahrenheit to Celsius converter as seen in Figure 2. While not all users may be comfortable with the backwards operation of a '×÷' or '+ -' component, we think that bidirectional data-flow can be an enriching feature when used at the right level. So we combine the overall data-flow behavior with a simple imperative scripting language inside of components. We used JavaScript as a scripting language but we are planning to integrate a visual tile scripting language such as Etoys [8], Scratch [10], or TileScript [18] to make the scripting part more end-user friendly.

Introducing an imperative language has its costs, the user has to cope with possible syntax errors and it violates the data-flow paradigm by possible access to global variables. By exposing the user to real source code only in a very narrow scope of *FunctionComponents*, many problems with textual languages are limited to the component where the user can fix it without having to browse a whole source code file.

3.5 User Interface

The basic user interface of Lively Fabrik consists of dragging and dropping components and connecting them afterwards. To fit the components better into the data-flow and to minimize crossings of connections the position of the pins can be changed by dragging them (as described in [9]). This leads to cleaner looking layouts. Other visual languages use fixed positions for connection points to associate spatial positions with functional behavior. To compensate this, we keep pins apart by varying color and potentially

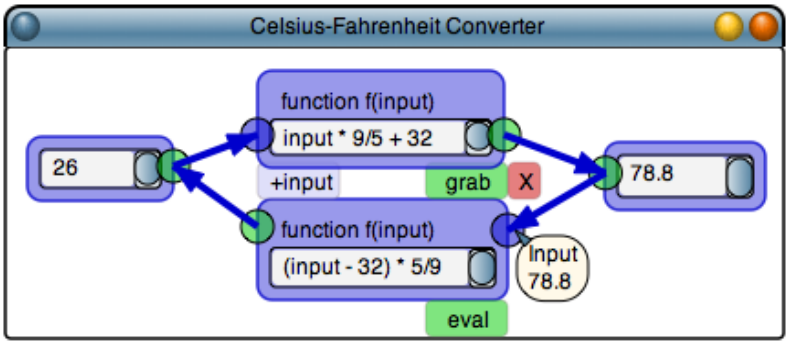

Figure 2. Fahrenheit Celsius Converter with *FunctionComponents* for each direction in Lively Fabrik

shape and display names in help balloons.

Bringing Together User Interface and Behavior In contrast to typical user interface builders and visual languages like LabView [3], Fabrik allows the user to create the user interface and the behavior in one place. This makes the programming more concrete, because there is one layer of indirection less in the system. *FunctionComponents*, pins, and connections can be hidden by specifying a region with a *UserFrame*. The *UserFrame* distinguishes elements that should be visible when the encapsulating component is reused. All components inside that *UserFrame* become the user interface for the encapsulating component.

Continuous Running Many end-user programming systems distinguish between creating programs and running them. So there is often a 'run' button that executes the program after its creation. This can lead to a long time where an initial program is created by the user but not executed. To solve this, dynamic systems often have a tight feedback loop where parts of the program or short snippets of code can be evaluated and tested, which makes things simpler and more interactive.

The use of a visual programming language allows us to go a step further, by manipulating the running program and objects directly, end-users can grow their program seeing the results of their changes instantly. This way of programming can occasionally frighten users, so we must assure, that while increasing the expressive power and possibilities of Lively Fabrik, users cannot accidently destroy their own work or valuable data. Having a browser and a Web-server between users and their data allows the integration of automatic versioning, which is not available in the interaction with files in an ordinary operating system.

Visual Aids The user interface of Lively Kernel inherits its directness and liveliness from Morphic [11], this enabled us to experiment with non standard user interfaces like halos as in Etoys [8]. We use these visual aids to show the state of the program to help users debugging their program. This should be further improved in the future, for example by displaying annotations of values, visualizing unused components, showing errors in the data-flow, or making visible how often a connection is used.

4 Implementation of Lively Fabrik

4.1 Architecture Overview

We separated the user interface and the domain model of our components as shown in Figure 4, because it allowed better development and automatic testing of the core component functionality.

Components, Models, and Morphs Components have a rectangular shape and contain user interface elements like text fields, buttons, and pins. We use NodeRecords as models, they store fields of values and object references and provide accessors and generate update events for registered observers. These update events make it easy to implement our data-flow model.

Morphs act as a view on models and components and are responsible for user interaction.

Pins and Connectors The data flows through connectors from pin to pin. This data-flow is implemented with an observer pattern that is provided by Lively Kernel. Pins and connectors use Morphs as their graphical representation and manage the observer relationships between component fields. Currently, connections are simple lines that have to be laid out manually but we plan to replace them by curves and add layout support, because the ease of use of connecting pins is very important.

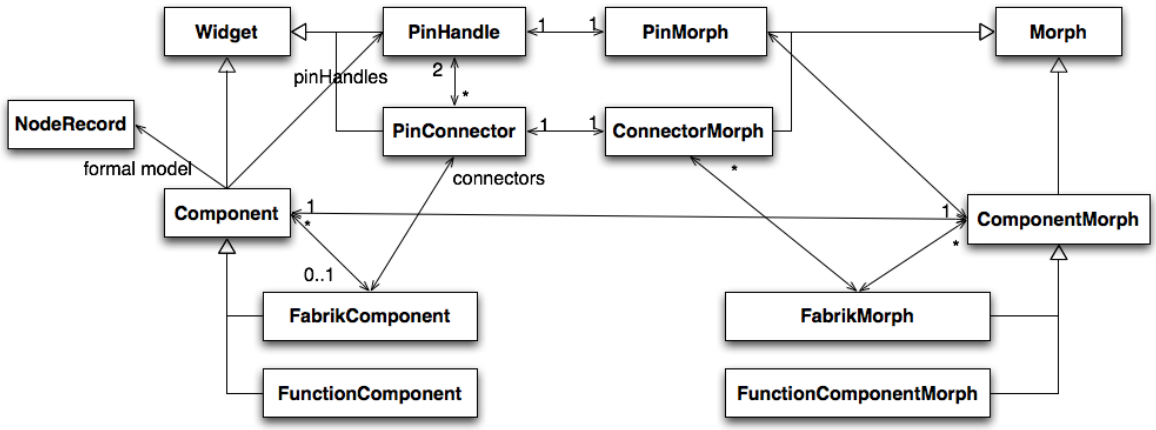

Figure 4. Lively Fabrik Core Architecture

4.2 Data-flow

There are many ways to coordinate the data-flow through a graph of components. Lively Fabrik does not have an overall rule but moves the responsibility to the component: each component decides for itself how to react on changes in pins. For example, a *TextComponent* displays a value, a *WebRequestComponent* retrieves content from the Web and a *FunctionComponent* evaluates its contents and produces an output. There can be cases when a *FunctionComponent* should wait for a set of new inputs, because they perhaps belong together, but for now we produce a new result for any change in input pins. Furthermore, there is no mechanism that stops circular loops in our implementation other than only propagating changes, when there is a real change.

4.3 Asynchronous Components

Components may not immediately produce an output to a given input. The *WebRequestComponents* take URLs as inputs and asynchronously perform a XMLHTTPRequest with the GET method. The response from the Web server is later written into the output pins. The output pin 'ResponseText' then gets a string version of the XML response assigned and the output pin ResponseXML gets a list of objects, each representing a XML tag of the response XML. Every of these objects has an attribute 'xml' and an attribute 'js', carrying a XML element respectively a recursively converted Javascript object of the XML element.

4.4 Integration with Lively Kernel

Lively Fabrik aims for a strong integration into the Lively Kernel to make graphics and widgets available that where not specially made for Fabrik.

Standard Lively Kernel widgets can be used as components in Lively Fabrik. They are wrapped by *PluggableComponents* which automatically generate pins from the model to provide an interface. An example for wrapped widgets is the *ClockMorph* that when dropped into Fabrik can act as source of ticking events to create animations.

For creating graphics, the Morphs of the Lively Kernel can be used. They can flow as references from one component to another, so that one component creates a Morph, the next sets some attributes and last moves it. The use of references violates the data-flow concept and collides with the change update mechanism, but it is a straightforward way to interact with the whole Lively Kernel.

Fabrik components can contain and encapsulate other components. This mechanism can be used to group behavior and abstract it into one component. Components inside a Fabrik can communicate to other components outside through the clear interface of pins. By copying the container component the behavior can be reused and shared.

4.5 Storing Fabrik Content

Lively Fabrik components are stored as other objects in the Lively Kernel as nodes in the DOM and serialized by the browser. Thus, not only main content such as the component types, their connections, and their current data is stored, but also their Morphs so that the user can customize the visual appearance without having to program. This may lead to conflicts in future versions, because the user may expect that his content updates itself in some ways and keeps the user changes in others.

Copy-and-paste of scripts for the purpose of customizing them is a crucial activity in end-user development. This is easy for text-based languages but difficult in a graphical environment, so visual scripts can only be shared by explicitly

importing whole projects or in the worst case by rebuilding everything from scratch according to a picture. By using the system clipboard and the serialization mechanism of the Lively Kernel, Fabrik components can be copied from one browser page to another.

5 Bringing It All Together

In the following, we describe the construction of a more complex Lively Fabrik application, a weather widget Mashup that queries a Web-service and has a nice user interface.

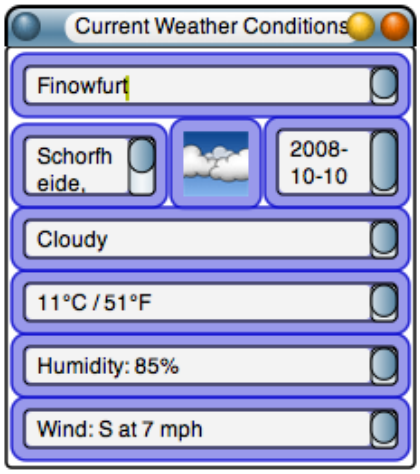

Figure 5. The user can enter the name of a city or a zip in the topmost input field to see the current weather conditions

Figure 5 shows the weather widget which allows users to type the name of a city or a zip code into the input field at the top of the component. The component then displays current weather conditions for the place as textual and graphical information below. This new component can be assembled by a user who is familiar with the basic Fabrik components. It can be saved inside a Lively Kernel world for publishing it in the Web. Other users are then able to use the widget, or if they intend to extend or change it, they can just 'uncollapse' the hidden parts of the application and modify them.

Basically, the construction of the weather widget consists of three parts:

- Requesting weather data via a Web service,
- Extracting and filtering this data from the result of the request,
- Creating a user interface for obtaining input and display output information.

To begin, a new *FabrikComponent* is dragged from the ComponentBox into the Lively Kernel world. Then a *WebRequestComponent* is dropped inside it. This component takes a URL as input and generates an XMLHTTPRequest using the HTTP GET method, thus retrieving data from the Web. For obtaining the weather data a Google service is used. The weather for Tokyo, for instance, can be retrieved by writing `http://www.google.com/ig/api?weather=Tokyo` into a *WebRequestComponent*. When the request is completed, the 'ResponseXML' pin of this component will carry a list of objects, each having two attributes: An attribute `xml` which points to an XML element (one for each tag) and an attribute `js` which is a JavaScript object generated from the XML element. It contains, recursively converted, all XML attributes and child nodes as ordinary object properties, thus allowing users to access the XML data without knowing anything about the DOM API or XML queries. When the 'ResponseXML' pin of the *WebRequestComponent* is connected to a 'List' pin of a *ListComponent* it shows a string representation of every element in the XML document.

The *ListComponent* now allows the selection of an object referencing sub elements of the XML tree which can be used in other components. Figure 6 shows the Web request part of the weather widget. Because the user should be able to pass in a city name or a zip, the URL will not be directly written into the *WebRequestComponent*. Instead, it will be produced by a *FunctionComponent* which concatenates the constant part of the URL with the city/zip. The result is the input for the *WebRequestComponent* which is then connected to two *ListComponents*. As described above, these are used to select parts of the provided information, namely the tags 'forcast_information' and 'current_conditions'.

To complete the request part, three new pins are added to the *FabrikComponent*: An input pin named 'Zip' which will be connected to the *FunctionComponent*, and two output pins 'Info' and 'Condition' which are connected with the *ListComponents*. When this is done, this part of the widget is finished and can be collapsed for hiding the internal components (the three pins of the *FabrikComponent* itself will not be hidden, they define the interface which can be accessed from the outside). To continue development, the *FabrikComponent* is embedded into another, newly created *FabrikComponent* by dragging and dropping.

In the new *FabrikComponent* (see Figure 7) the components used for the user interface will be added. To get started a *TextComponent* is necessary which will take the city/zip. It will be connected to the 'Zip' pin of the embedded *FabrikComponent*. When a valid city/zip is entered in the *TextComponent* the selected results of the request will appear on the 'Info' and 'Condition' pins of the embedded *FabrikComponent*.

As mentioned above, these are objects that contain se-

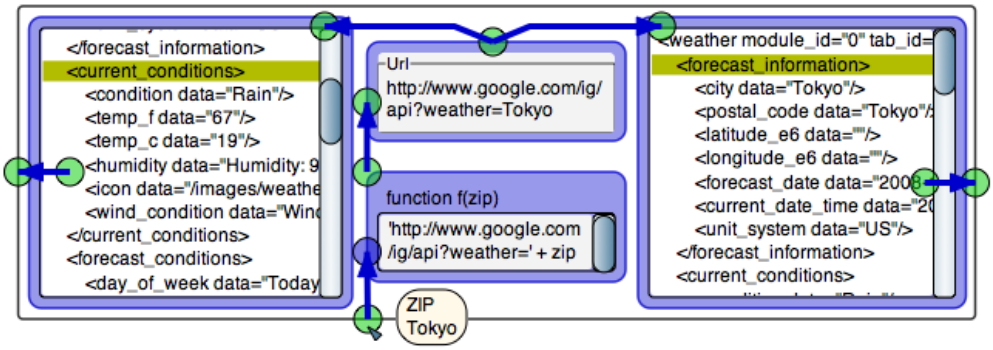

Figure 6. Requesting data from a URL and extracting information from the received XML

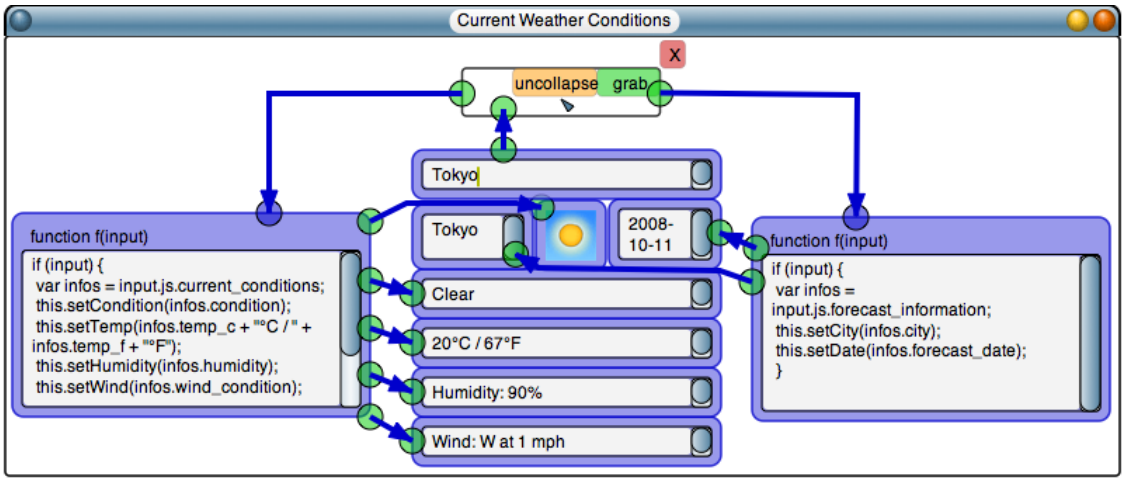

Figure 7. The complete Weather component. For simplification the component from Figure 6 is collapsed.

lected XML data, both the raw XML as well as a JavaScript object generated from it. To get, for example, the Celsius temperature, the user can connect the 'Selection' pin of the *ListComponent* containing the selected 'current_condition' tag to the 'Input' pin of a *FunctionComponent*. The input variable then references the object bearing the converted XML data.

To avoid creating a *FunctionComponent* for each *TextComponent*, multiple outputs from one *FunctionComponent* are realized by creating the required number of output pins and then setting the values for those pins using JavaScript inside the 'FunctionBody'.

Now all components are created and connected. A *UserFrame* is drawn around the *TextComponents* and *ImageComponents* simply by clicking in an empty area of the *FabrikComponent* and dragging the frame so that it contains all components which should be seen by a user. As shown in Figure 5, when the *FabrikComponent* is collapsed all components that are inside the user frame will remain visible, but their connections and pins are removed from the view.

This example showed the construction of a more advanced Lively Fabrik application. New components are created by connecting existing components graphically and writing small scripts. All of this is done at run-time. This means that interim results can easily be debugged during construction by inspecting pin values or adding *FunctionComponents* and using JavaScript for debugging. This allows the quick detection of bugs and enables users to rapidly evolve Lively Fabrik applications. Due to this simplicity and the effortless creation of new abstraction we believe that Lively Fabrik is capable of creating much more complex systems than the one described.

6 Related Work

Lively Fabrik is named after the experimental interactive programming environment Fabrik [6, 9]. In Fabrik programming was done by placing components in a graph and

connecting them with wires. It integrates the programming process and the creation of the user interface in one environment. Fabrik uses the structure data-flow model and thus is timeless and allows connections to be potentially bidirectional. User interface elements support user input and the program cannot only print results, but generate graphics, for example for creating diagrams. The original Fabrik is implemented in Smalltalk and uses code generation to speed things up. Lively Fabrik's *FunctionComponents* directly create JavaScript functions that can be evaluated without generated classed or a graphical representation.

Many visual programming languages use the metaphor of data-flow, popular representatives are *LabView* [3] and the Visual Language in *Microsoft's Robotic Studio* [14]. An overview of current visual data-flow languages is given in [7].

Another approach to end-user development of Web applications is described in [16]. Rode et al. surveyed casual Web masters without programming knowledge as their group of end-users. The resulting tool *phpClick* is meant for basic data collection, storage and retrieval applications. Mashups and more graphical Web applications are not in their focus.

There are browser-based products such as *Yahoo Pipes* [19] and *Microsoft Popfly* [13] that allow users to generate Mashups, but they are restricted in their ability to integrate the results into Web applications. Yahoo Pipes generates restricted views to display the output and provides the result in form of feeds for reuse in other contexts. Popfly lets users integrate created widgets into their own Web applications (via copy and paste JavaScript code, and other ways) but there are few end-user Web environments for creating general Web applications.

The data-flow paradigm is not only used in end-user environments for creating Mashups, but it has a also an application in operating system scripting. *Apple Automator* [1] is an end-user development tool for automating applications and the Macintosh operating system. It uses a pipe metaphor in which data flows like in pipe from the top to the bottom from one action to the next (actions correspond to our components). In this mainly linear flow loops and variables are possible, but they don't fit in naturally into the system. Some Automator actions provide access to Unix shell scripting and AppleScript [4]. This combination of overall data-flow with scripting inside of actions makes it a very open and powerful system.

7 Summary and Outlook

We have designed and implemented Lively Fabrik, a Web-based end-user programming environment. To make system development more appealing to end-users, Lively Fabrik combines the data-flow of visually wired components with dynamic scripting capabilities. In Lively Fabrik, end-users do not have to split their work on the application's core behavior from the associated graphical user interfaces, since these can be easily and automatically separated if needed. With Lively Fabrik being part of a client-side interactive Web application environment, there is no need to install or update a development environment, since end-users have instant access to an always current and rich programming substrate.

As of today, Lively Fabrik has only basic support for the collaborative development of Web applications. We plan to extend these capabilities to experiment with near real-time collaboration in a widely distributed Web-scale environment. Lively Kernel's entire functionality can be made accessible through our *FunctionComponents*. To improve modularity and with that comprehensibility of more complex applications, we will provide means similar to procedural abstraction to group, extract, and offer abstractions meaningful and useful in particular programming situations. Examples are Web content extraction, filtering, syndication, and the provisioning of graphical objects for direct interaction.

Integrating tile scripting languages like *Etoys* [8] or *Scratch* [10] seems to be a promising alternative to JavaScript in *FunctionComponents*. *TileScript* [18] is a convincing example of how a textual scripting language like JavaScript can be transformed into a visual representation and back. Combined with the graphical capabilities of Lively Kernel, this may result in an Etoys-like system in the browser.

We are working on improving the usability of Lively Fabrik based on feedback from actual end-users when applying our programming environment to creatively express their ideas and solve their problems.

8 Acknowledgements

We thank Krzysztof Palacz for fruitful discussions, valuable contributions, and his creative work on Lively Kernel, and Philipp Engelhard for for his comments on an early draft of this paper.

References

[1] Apple. Automator: Doing Things Over and Over is Over, 2007. as of Sep 23 2008, `http://automator.us`.

[2] E. Baroth and C. Hartsough. Visual Programming in the Real World. pages 21–42, 1995.

[3] R. Bitter, T. Mohiuddin, and M. Nawrocki. *LabVIEW: Advanced Programming Techniques*. CRC Press, 2006.

[4] W. R. Cook. Applescript. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 1–1–1–21, New York, NY, USA, 2007. ACM.

[5] D. Ingalls, T. Mikkonen, K. Palacz, and A. Taivalsaari. Sun Labs Lively Kernel, 2007. as of Oct 12, 2007, http://research.sun.com/projects/lively/.

[6] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik: a visual programming environment. *SIGPLAN Not.*, 23(11):176–190, 1988.

[7] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[8] A. Kay. Squeak Etoys Authoring and Media, 2005. as of Aug 01, 2005, http://www.squeakland.org/pdf/etoys_n_authoring.pdf.

[9] F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace, and K. Doyle. The Fabrik Programming Environment. *Visual Languages, 1988., IEEE Workshop on*, pages 222–230, Oct 1988.

[10] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A Sneak Preview. In *C5 '04: Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. H. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, New York, NY, USA, 1995. ACM.

[12] D. Merrill. Mashups: The new breed of Web app. *IBM Web Architecture Technical Library*, 2006.

[13] Microsoft. Popfly, 2008. as of Sep 23 2008, http://www.popfly.com.

[14] S. Morgan. *Programming Microsoft Robotics Studio*. Microsoft Press, Redmond, WA, USA, 2008.

[15] Y. Ohshima, T. Yamamiya, S. Wallace, and A. Raab. TinLizzie WysiWiki and WikiPhone: Alternative approaches to asynchronous and synchronous collaboration on the Web. In *C5 '07: Proceedings of the Fifth International Conference on Creating, Connecting and Collaborating through Computing*, pages 36–46, Washington, DC, USA, 2007. IEEE Computer Society.

[16] J. Rode, M. B. Rosson, and M. A. P. Quiñones. End User Development of Web Applications. In F. P. Henry Lieberman and V. Wulf, editors, *End User Development*, volume 9 of *Human-Computer Interaction Series*, pages 161–182. Springer, 2006.

[17] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web Browser as an Application Platform: The Lively Kernel Experience. Technical Report SMLI TR-2008-175, Sun Microsystems, January 2008.

[18] A. Warth, T. Yamamiya, Y. Ohshima, and S. Wallace. Toward A More Scalable End-User Scripting Language. *Creating, Connecting and Collaborating through Computing, 2008. C5 2008. Sixth International Conference on*, pages 172–178, Jan. 2008.

[19] Yahoo. Pipes, 2008. as of Sep 23 2008, http://pipes.yahoo.com/pipes/.