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. It supports its users in creating dynamic Web content/pages from within their Web browsers. Based on the Lively Kernel, Lively Fabrik extends the ideas of the original Fabrik system by empowering end-users to entirely compose Web application elements and their associated behaviors into interactive Web experiences. Web applications created with Lively Fabrik typically combine Meshups of Web sources, data manipulation, and interactive user interface elements, but can be, 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 [16]. Besides reading the Web, they manage their personal correspondence via email, contribute to Wikipedia articles, collaborate in online spreadsheets, or just meet with their friends. The creation of Meshups [11] by combining Web-based services from different sources into a single application dedicated to a particular task, and the customization of existing Web applications or content management systems [REF NEEDED] are other examples of end-user control that have become popular.

Even though development support in these areas benefits from restricted domains, usability and simplicity offered to the end-users are still lacking. One of the reasons why Web-based applications are inferior 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 environments built on Lively Kernel [4], and based on Fabrik [5]. Lively Fabrik is an environment that empowers end-users to interactively create their own dynamic Web pages from within their own 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 as simple as possible. 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 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 to help end-users structuring their programs with the data-flow paradigm and provide scripting for cases where the use of data-flow is complicated or limits the end-user.

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 [**?**] and sharing of content.

The remainder of the paper is organized as follows section 2 describes the visual web-based end-user programming environment Lively Fabrik. Section 3 discusses implementation details. Section 4 shows the usage of the environment by creating a weather widget Meshup. Section 5

discusses related work and Section 6 makes a summary and gives and 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. This 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.

Web-based programming is different from programming on personal computer. Programmers have to cope with client/server aspects, security limitations, less CPU cycles to burn and inferior libraries. For example because of security reasons the clients side has no access to local files and services. Some of these issues are addressed by modern browser technologies like faster JavaScript virtual machines and more powerful browser APIs make new application programming environments like the Lively Kernel possible. Other issues like the security limitations have to be dealt individually. JavaScript applications that need access to the internet, as need a web proxy on the server the original JavaScript source file is from.

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.

All these online applications are limited when it comes to programming in the browser and specifically programming for end-users in the browser.

An important domain of end-user programming in the browser is the creation of Meshups. A Meshup (for an overview see [11]) is a recombination of different Web pages or services in a new and often unanticipated way. It consist typically of a script running on a server, that fetches data from different sources and produces new Web content.

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

Lively Fabrik brings the ideas of Fabrik [5, 8] 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.

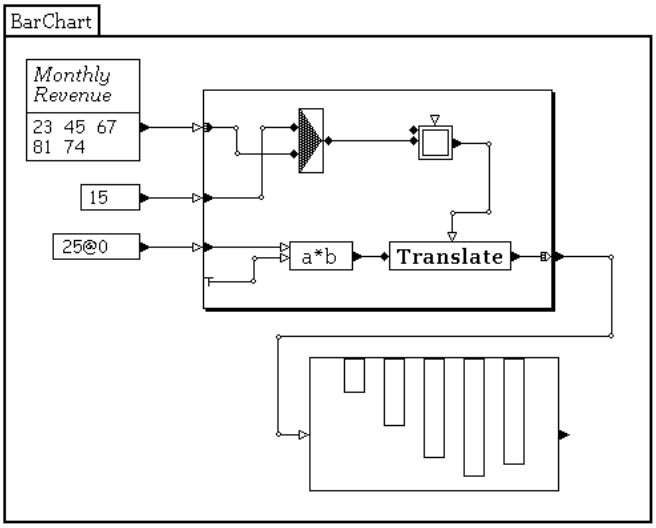

Figure 1. Bar chart in the original Fabrik [5]

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.

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 components that are connected with wires, through which data flows.

Components, Pins, and Connectors 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 a connections. Lively Fabrik defines a set of primitive components which are not created in Fabrik itself. These components are:

- TextComponents display strings which can be both user input or data from other components.
- 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' and a pin 'Result' which automatically gets the return value of the JavaScript code assigned.
- 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.
- 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.
- WebRequestComponents take URLs as inputs and asynchronously perform a XMLHTTPRequest with the GET method. 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.
- FabrikComponents provide a container 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 fram are not hidden. With FabrikComponents users are empowered to create their own components.

* pins are the the interface to components When the user wants to connect two components, he clicks 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). The semantic of the connection is when the value in the pin of the first component changes, it is transported to the other pin. Thus the connection is directed. Bidirectional connections are created by reverse connecting two pins. Some components trigger actions when a pin value changes (a FunctionComponent executes its associated function, a WebRequestComponent triggers a XMLHTTPRequest and so on).

Behavior of Components The response of a component to changes in pin values may be predefined, for example a text component displays the value, or a URL component retrieves content from the Web. The other possibility is that the component is scriptable or that it encapsulates other components.

3.2 Visual Data-Flow

Wiring components is visual data flow programming. We have chosen that paradigm because it is intuitive and most users think in terms of data going from one place to another [2, 6].

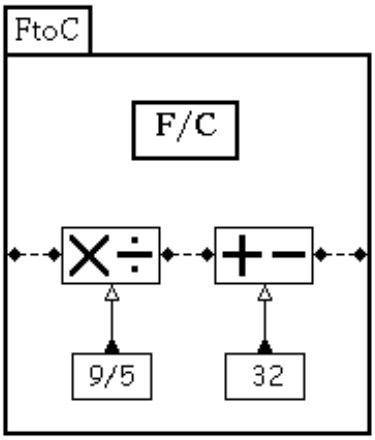

Figure 3. Bidirectional Fahrenheit Celsius converter in the original Fabrik

There are many ways to coordinate the data flow through a graph of components. The original Fabrik uses a topological sort to decide with what input parameters a component produces new output parameters. 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 input pins.

Lively Fabrik does not restrict the type of data that flows through the wire. 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.

3.3 Enhancing the data flow principle with scripting

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

The original Fabrik demonstrated bidirectional data flow with an 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 dataflow 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. As a scripting language we used JavaScript but or visual tile scripting language like Etoys [7], Scratch [9] or TileScript [17] could also be taken. Introducing an

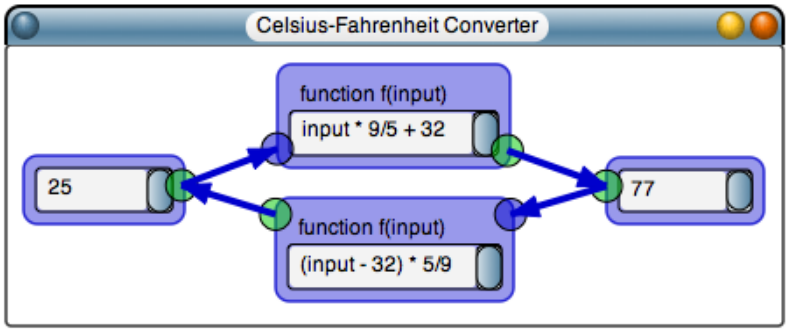

Figure 2. Fahrenheit Celsius Converter with Function Components for each direction

imperative language has its costs. By exposing the user to real source code only in a very narrow scope of function body, problems with textual languages like syntax errors are limited to that component where the user can fix it without have to browse a whole source code file.

3.4 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 dataflow and to minimize crossings of connections the position of the pins can be changed by dragging them (as described in [8]). This leads to much cleaner looking layouts. Other visual languages use fixed positions for connection points to associate spacial positions with functional behavior. To compensate this, we keep pins apart by varying color and potentially 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. To distinguish elements that should be visible when the encapsulating component is reused, function components, pins and connections can be hidden by specifying a region with a 'User Frame'. All components inside that 'User frame' are 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 much more simpler and 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 the user. So we must make sure that, while increasing the expressive power and possibilities of Lively Fabrik, we also secure user data and program itself so that the user can not destroy his own work or valuable data. Having a browser and a web-server between the user and his data allows the integration of automatic versioning, that is not available in the interaction with files in an ordinary operating system.

Visual Aids The user interface in Lively Kernel inherits its directness and liveliness from Morphic [10], this enabled use to experiment with non standard use user interfaces like halos in etoys [7] or custom balloon helps. We use these visual aids to show the state of the program to help the user debugging its program. This should be further improved in the future, for example by displaying annotations of values, visualizing unused components or connectors, errors in the data flow, or how often a connection is used.

3.5 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.

Pluggable Components Standard Lively Kernel widgets can be used as components in Lively Fabrik. Pins are automatically generated from the Model of the widget and the Morph is embedded as a pluggable component. An example for wrapped widgets is the Clock Morph that can act as source of ticking events when dropped into Fabrik.

Graphics 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 user sets some attributes and the third moves it. The use of references violates the data flow concept and collides with the change update mechanism, but it is a straight forward way to interact with the whole Lively Kernel.

Encapsulating Components Some components can contain and encapsulate other components. This mechanism can be used to group behavior and abstract it into one component. The components inside can communicate to other components outside through pins of the container component, so that the interface is clear. By copying the container component the behavior can be reused and shared.

4 Implementation of Lively Fabrik

4.1 Architecture Overview

Lively Fabrik is about visually wiring together components. We separated the user interface and the domain model of our components as shown in Figure 4. Components glue together models and Morphs as their visual appearance.

4.2 Components, Models, Morphs

Components have typically a rectangular shape and user interface elements like a text field and pins. The pins, provide an interace to its value fields. Components can react on value changes in pins and put new values out to their own pins.

A model is a collection of fields of values and object references. It provides accessors and generates update events for registered observers and is responsible for persisting the state of a component.

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

4.3 Pins and Connectors

The data flows through connectors from pin to pin. This data-flow is implemented with an observer pattern, which 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.

4.4 Storing Fabrik Content

Lively Fabrik components are stored as any other objects in the Lively Kernel as DOM elements of a Web page. Thus not only the real content such as the component types, their connections, and their current data is stored, but also their user interface 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 and 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

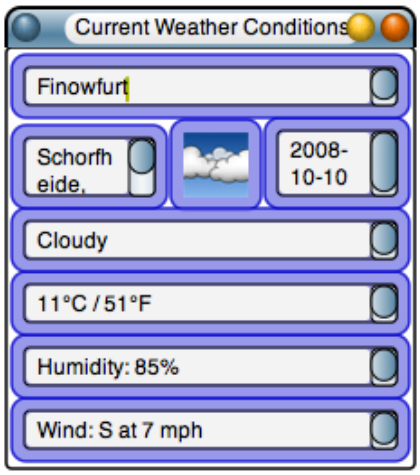

Figure 5. The user can enter the name of a city or a zip in the topmost input field to see the current weather conditions

Figure 5 shows a more advanced Fabrik application. This application 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 informations below. This new component can be assembled by a user knowing the basic Fabrik components and how to use them in about 10 minutes. It can be saved inside a Lively Kernel world for making it available. Other users are then able to use the

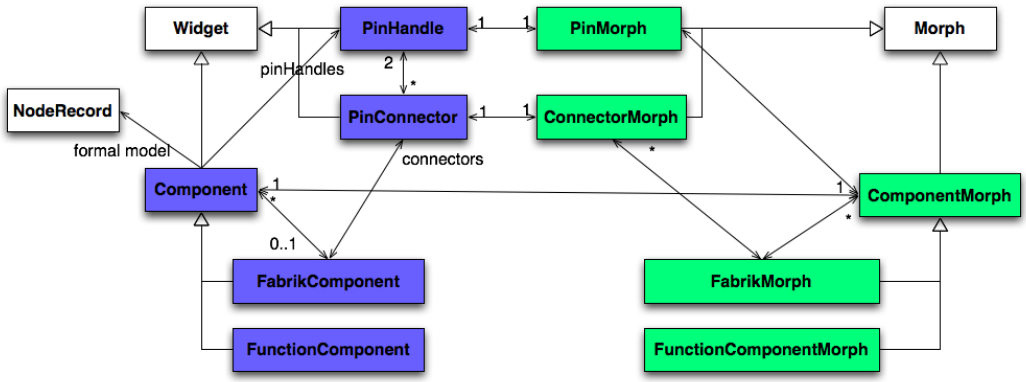

Figure 4. Lively Fabrik Core Architecture

widget, or if they intend to extend or change it, they can just 'uncollapse' the hidden parts of the application and modify them.

The remainder of this section describes the creation of the weather component.

Basically, the construction of the Weather Component consists of three parts:

- Requesting weather data via a Webservice,
- Extracting/Filtering this data from the result of the request,
- Creating a user interface for obtaining input and display output informations.

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 we use a Google service. To get the weather of Tokyo, for instance, simply write http://www.google.com/ig/api?weather=Tokyo into a WebRequestComponent and accept it. 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 is pointing 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 Component. 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 ZipOrCity 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 ZipOrCity 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 which contain selected 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

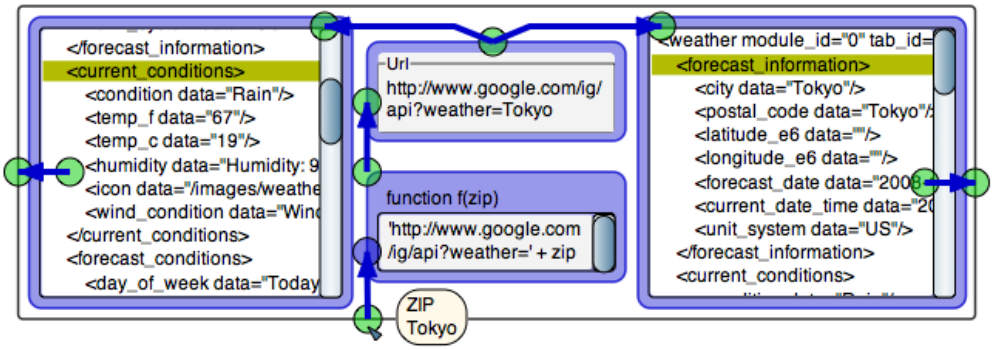

Figure 6. Requesting data from a URL and extracting informations from the received XML

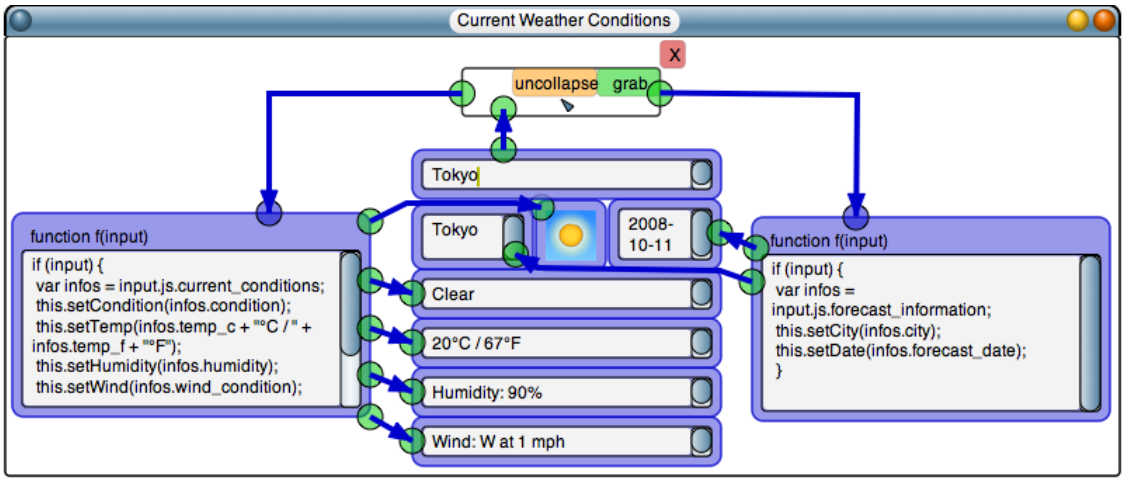

Figure 7. The complete Weather component. For simplification the component from figure ?? is collapsed.

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 user frame is drawn around the Text/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 all components which are inside the user frame will remain visible when the FabrikComponent is collapsed, but their connections and pins are removed.

6 Related Work

Lively Fabrik is named after the experimental interactive programming environment Fabrik [5, 8]. In Fabrik Components placed in graph and wired by connecting pins. It integrates the programming processe and 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 can not 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 Fabriks FunctionComponents directly create JavaScript functions that can be evaluated without generated classed and methods and the graphical representation.

Many visual programming languages use the metaphor of data flow, popular representatives are LabView [3] or the Visual Language in Microsoft's Robotic Studio [13]. An overview of current visual data flow languages is given in [6].

Building Web applications in the browser is also a topic in the Pier CMS [14], where the user can specify the be-

havior of their Web pages reusing ... TODO: read and then write

Another approach to end-user Development of Web Applications is described in [15]. 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. Meshups and Active Object are not in their focus.

There are browser based products as Yahoo Pipes [18] or Microsoft Popfly [12] that allow users to generate Meshups, but they are restricted in their ability to integrate the results into Web applications. Yahoo Pipes generates very 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 dataflow paradigm is not only used in end-user environments for creating Meshups, but it has a recent application in operating system scripting, too. Apple Automator [1] is end-user development tool for automating work with applications and the operating system. It uses a pipe metaphor where data flows from the top to the bottom from one pipe to anther. It 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 apple script which makes it a very open and powerful system, that combines overall dataflow with scripting inside of components (here called actions).

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, since end-users have instant access to an always current and rich programming substrate.

As of today, Lively Fabrik has only very 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 function components. 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 here are Web content extraction, filtering, syndication, and the provisioning of graphical objects for direct interaction.

Integrating tile scripting languages like those of Etoys [7] or Scratch [9] is an promising alternative to using JavaScript in function components. TileScript [17] is an interesting example of how a textual scripting language such as 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, and studying 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 we would like to thank 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.

[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] 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/`.

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

[6] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[7] A. Kay. Squeak etoys authoring and media, 2005. as of Aug 01, 2005, `http://www.squeakland.org/pdf/etoys_n_authoring.pdf`.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] L. Renggli. Pier–The Meta-Described Content Management System ESUG Innovation Technology Awards 2007. Technical report, University of Bern, Mar 2008.

[15] 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.

[16] 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.

[17] 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.

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