ServiceDesigner: a Tool to Help End-Users Become Individual Service Providers

Fredrik Espinoza and Ola Hamfors
Swedish Institute of Computer Science
Box 1263, 164 29 Kista, Sweden
{espinoza, hamfors}@sics.se

Abstract

Pervasive and ubiquitous computing will fundamentally change the mode of interaction between humans and computers; instead of working with applications on a desktop computer we will interact with our environment through electronic services—anytime and anywhere. In this new modus operandi, specialized and personalized services will become much more important; the usual software house solutions may not be sufficient for individual demands. We propose that end-users themselves can be the service providers; the incentive to create services is grounded in each individual's personal demand for well suited services and this demand will only increase when technology makes it possible to access services ubiquitously. Individual Service Provisioning requires three parts: a general platform for managing and accessing electronic services; simple but powerful tools to create the services; and the means to share services between users. Building on previous work developing sView, a personal service environment, this paper presents the second part—ServiceDesigner—a tool for creating new services for sView. ServiceDesigner, using web services that expose the functions of web sites as programmatically accessible components, lets end-users create personalized and functional electronic services that fit in the personal platform. With ServiceDesigner, web services are directly available to users and finished services can also be shared with other users.

1. Introduction

With the Java model, any and everyone can be a service provider.1 We believe computer usage is moving toward a pervasive, mobile, and service centric model. The World Wide Web is evolving from publication of information to provision of interactive services, while mobile communication and computing, including mobile phones and PDAs, are gaining a wider acceptance alongside a wider deployment of interactive services over broadband, cable-TV, and digital-TV networks. In a somewhat more distant future the advancement will move even further, toward a true ubiquitous computing model, wherein traditional computers may disappear completely, to be replaced by intelligent artifacts powered by highly specialized computing devices.

Interactive electronic services and their deployment, delivery, and user access, constitute the main topics of interest for our ongoing sView project2 [8, 9]. With the sView system, each user gets a personal service briefcase in which to put their personal electronic services. The briefcase, and the placement of the user’s services therein, improves the user’s control over services, enables services to adapt to the user and to access-devices, enables services to cooperate, and—with the ServiceDesigner tool—enables users to become individual service providers.

ServiceDesigner is an sView service that lets end-users without special programming skills create new services for sView. In a visual editing environment, the user takes one or more Web Services (programmatically accessible and networked functional components), and designs a graphical user interface to the new electronic service. The resulting service is compatible with the sView service environment, and as a self-contained JAR file (Java ARchive—a compressed format for Java files), it may be shared with other users.

Services, in general, may be categorized as being built from scratch, being constructed from a template, being combined from components, or a combination of these.

• Services that a professional developer creates from scratch. This type of service requires significant technical skill on the part of the provider. Much of the

---

1Scott McNealy, founder and CEO of Sun Microsystems Inc., the Java ONE Conference, San Francisco, California, March 2002.

2SView is a project and a system in the OASIS group of the Swedish Institute of Computer Science.
work of creating a service is specific to the particular service and the platform for which it is created
- Services constructed from templates. These services are made of pre-constructed service shells that the provider fills with his valuable content to make up the service
- Combinations of building blocks. These are services that the provider has built by combining other services and connecting them to create new functionality
- Combinations of the previous

With the ServiceDesigner and the advent of web services, new combinations of services, as in the third type above, will be within reach of end-users; the notion of every user being a service provider is set to become reality. The number of professionally created services for the pervasive and ubiquitous environment will be great but the number of individually created services will be even greater.

1.1. Motivation

The motivation for ServiceDesigner stems from a practical level and a higher more conceptual level.

On a practical level, ServiceDesigner enables users to easily\(^3\) access and directly make use of web services, and to use web services as building blocks when creating new services.

On a high level, we aim for an Open Service World\(^4\), in which a user is able to access highly specialized services to fill the needs of any particular moment. Since individual users’ needs will differ, either services must have the ability to adapt, or services must be tailored to suit individual users or smaller clusters of users. Large software houses may be unable to adapt and quickly satisfy these needs. This implies an opportunity for small and medium size companies to fill the void, since these companies may be more flexible and better equipped to adapt to users’ demands. However, even the more flexible service provisioning of smaller actors may not be enough. If an appropriate service does not exist, just as anyone can publish a web page about themselves or their interests or business, anyone should be able to create a service for themselves or others to use. We call this Individual Service Provisioning.

To accomplish this we need three different systems:

- The first part is the personal service platform sView \[^8\], which provides a personal service environment for each user. We believe that such as system is necessary to mitigate the open world of services; without it, the usage of services becomes burdensome, and, concomitantly, the incentive to create services will falter
- The second part, ServiceDesigner, empowers end-users to create their own services
- The third part, a peer-to-peer system called BriefcaseConnectivity \[^11\], provides a generic networking system for services in sView and enables services (including services created by users with ServiceDesigner) to propagate among users in the sView community. More services means more benefit to users and acts as positive feedback to encourage overall usage

The ServiceDesigner is described in the current paper, the other systems are outside this scope and are described elsewhere (e.g. \[^9, 8, 18\] and \[^11\]).

We are also motivated by trying to answer the following questions:

- How do we stimulate the production of a greater range of special purpose and individualized services?
- How can we enable individuals to easily create new service combinations?
- How can we make it possible for unrelated services to come together to provide services above and beyond what they are capable of individually?

1.2. Contribution

ServiceDesigner allows end-users to easily and quickly test web services and to create new services for sView. It allows the user to both create and utilize a single web service, or to connect several web services to create a new service. A graphical user interface, a web-based interface, and a Wireless Application Protocol (WAP) interface are automatically generated for the resulting service ensuring fast and easy access in many mobile environments.

1.3. Limitations

Please consider the following limitations and caveats when reading this paper:

- It should be evident that there is potential for a great number of services to be available to an sView user. Even if the future platform of choice is not sView, the same types of services will most likely be sought after and therefore produced, albeit for other platforms. Thus, regardless of platform, the reasoning that follows should be applicable. In this text, for reasons of familiarity, we choose to consider sView.
- Surely, Individual Service Provisioning also requires an appropriate market model including payment methods that are flexible enough to allow for the types of
payments and the amounts that individual service provisioning requires. This, however, is beyond the scope of the current paper.

- There are many ways to describe networked services. We limit ourselves by choosing Web Services Description Language (WSDL) [19] because it seems likely that it will become an accepted standard. WSDL is a rich language and it enables you to describe communication of complex objects. For simplicity, we decided not to support complex object communication, limiting the system to communication of simple data types like integers, strings, etc.
- There are many communication protocols for making “remote procedure calls” (RPC). We decided to use Simple Object Access Protocol (SOAP) [17], because it uses HTTP as a base transmission protocol and therefore is excellent when communicating over the web.
- Within the Semantic Web research there are efforts similar to the present work [3, 15, 13, 6, 2]. The focus of the web service related work, however, is often on automatic and semantic matching of services to each other. The focus and approach of the present work is on the manual, human assisted matching of services to each other, complemented by sharing of created services. We believe that the two approaches aim for the same goal and that they are complementary.

2. Background

The World Wide Web has evolved from a system for publishing static information in the form of web pages, to providing more and more advanced and interactive services. Unfortunately, the web was not designed for this type of usage, which is evident in, first, the proliferation of add-on protocols and functionality that have been added to the original web infrastructure to keep it running smoothly despite its new requirements, and, second, the advent of web services. Consequently, and in response to the evolution of the web and the growing importance of other domains of electronic services, we designed and developed sView [8, 7, 9]. The following sub-sections briefly describe sView and web services.

2.1. SView

The idea behind the sView system is to collect all personal services in one place, a personal service briefcase, which is available to the user at all times and from all platforms. The sView platform, with its user centric focus, can alleviate some of the problems of the web as a platform for electronic services and at the same time provide a more flexible and open environment for the future.

The services are kept as close to the user as possible, since local access to services means that we can provide the best possible mode of interaction. Thus, when the briefcase is running on the user’s workstation at work or on a PC at home, the user may access services using the graphical user interface. When the user logs out, the services are automatically transferred to a sView Enterprise server in the network, and the user may continue their access using a remote interface such as a mobile phone or a web browser. As long as a user keeps services in their briefcase they are constantly active, even if the user logs out of the system.

There are more benefits that stem from keeping all services in the same environment. First, some services in the briefcase can be used by other services. These generic provider services expand the functionality of the user’s briefcase. At the same time they allow service providers to focus on the core business of their services since they can use the shared provider services for functions that are common to many services. Graphical user interface capabilities and peer-to-peer network support has been implemented in this manner.

Next, the user has a greater degree of control over his services when they are all in the same environment. First, the user’s personal preferences may be exposed to services as the user chooses and under the user’s supervision. All the user preferences can be stored in a preference service to which other services are allowed to connect and retrieve data. Since the preferences are kept in one place, updates are made in one place and then propagated to the necessary services. For example, if the user changes his address he only needs to do it once, in the preference service. Second, with digital signatures and certificates, the user can guarantee the validity and source of a service. Sview has a built in security system with which an un-trusted service can be isolated from the others to preserve the briefcase integrity.

Finally, services can cooperate. This is possibly the most far reaching benefit of sView. The provider service scheme of sharing common functionality is the principal method of service cooperation in sView. During the creation of a service, the service provider may implicitly link to provider services that will be required in the briefcase for the service to function properly. However, there is also a way to make services cooperate dynamically. We describe these concepts in greater detail below, in section 3.2 (Connecting Services).

2.2. Web Services

Web services are one type of electronic services. They are modular and programmatically accessible networked components based on XML descriptions and deployed using existing web infrastructure. This new generation of networked electronic services will be available to software de-
velopers as distributed pieces of functionality that can be incorporated in new applications as they are being developed. Web services are, however, not being targeted at end-users. Since they inherently have no user interface, only the programmatically accessible interface, they must be incorporated into an application before they can be indirectly used by end-users.

Network calls to web services are often performed using SOAP over HTTP and the services are described using WSDL (although other configurations are possible).

Using SOAP, you literally include a standardized XML-element into a HTTP message where the SOAP specification defines the structure of this XML-element. The XML contains information that enables client/server communication but not as advanced as in the distributed object models. And that is why SOAP is not a replacement for the complex object models - SOAP is a complement, intended to be used when communicating over the web.

A WSDL document describes operations as a set of endpoints that can process information. These operations are described in an abstract way, to isolate them from specifications of the kind of communication protocols and formats that are used. The parameters that the operation takes are described with XML-schemas. The document contains all information required to make a callable instance of the service:

- Where the operation is located on the network
- What format is required for the message
- Which protocol should be used when sending the message
- A description of the required parameters
- A description of the response to expect

SOAP and WSDL are key to creating networked services that are programmatically accessible to others.

3. ServiceDesigner

We now take a closer look at the ServiceDesigner system.

The ServiceDesigner should assist the user in the following steps:

- The first step is to let a user access web based service descriptions (WSDL documents). The user should be able to get information on what kind of functionality the service provides (i.e. its functional components) and then be able to choose which functionality to use. It should be possible to choose functional components from several different web services
- The next step is to automatically generate a default graphical user interface for the chosen functionality. The user should be able to modify and design the graphical user interface with a visual tool. The service should be as accessible as possible, i.e. have several different graphical interfaces like HTML, WML, JAVA-SWING etc. As soon as the user interface has been generated it should be possible to test the service with real parameters
- In the last step the designed user interface together with a specification of the functional components that have been used is combined into an sView compatible service. This step involves integrating code that is necessary to display the interface, make the SOAP function calls, and to function as an sView service. The integration results in an sView service that is ready to be loaded into the personal service briefcase or distributed to other users
- We also complement the basic functionality described above with support for connecting together several functional components in one coherent service. The user should have some (preferably total) control over the information flow between the parts of functionality. We describe this further below, in section 3.2.

Another design goal was to make the ServiceDesigner easy to use. This means that an end-user should be able to use it without writing any code. The ServiceDesigner is designed with these criteria in mind.

3.1. Creating a Basic Service

The first step in creating a new service is to load the service descriptions of the component parts (see Fig. 1). The URL of the WSDL document is entered into the top text field of the main window of ServiceDesigner. When the “Ok” button is clicked, the WSDL document is loaded and a list of available functional components is displayed in the “Available Functionality” area. The user can choose from this list and add functional components to the “Added Functionality” area below. The chosen functional components will be included in the service and its interface when the user clicks the “Next” button to go to the next step, generating the interface.

3.1.1 Generating the Interface

When the interface is generated each functional component is represented by a set of text fields and labels, and an activation button. In Fig. 2 we see that one functional component was chosen.

The text fields with the corresponding labels represent input parameters to the functional component. The labels are extracted from the WSDL document and tend to be more or less descriptive. The activation button is generated in such a way that clicking it activates the function with the contents of the text fields as parameters. There is also a
Figure 1. The main interface window of the ServiceDesigner. In this interface the user inputs a URL to a service description document which is loaded and understood by the system. The user then chooses from a list of available functional components (items in the middle area).

The default generated graphical user interface can be modified by the user, which means that the user can change the default graphical components to something more appropriate. A text field may be changed to a text area for more space or to a selection box (pop-up menu) for pre-defined values. The user can also set the value of a text field to be final, which implies that the same value will always be used.

Moreover, label texts can be edited, positions and sizes of components can be changed, and the output area size and position can be modified.

3.1.2 Generating the sView Service

Finally, when the user is happy with the design of the interface, the sView service can be generated. The user chooses File/Generate Service and then fills in a dialog asking for a name of the service, the authors name, keywords, and any other text that can help a potential future user to understand and make use of the new service.

We chose to generate the service as JAVA-source code that can be compiled with generated scripts. The scripts also generate a JAR file version of the service. This JAR file, which can be distributed to other users, contains everything that is needed in order to run the service in sView.

After the service has been generated it is ready to be used and it is automatically loaded into sView.

3.2. Connecting Services

One of the most significant properties of the ServiceDesigner is its ability to connect together several different functional components from several different web services. This means that completely unrelated services can be combined into never before seen combinations simply according to the needs and taste of the user. When such a combined service is created it too may be shared with others, and just as in the case of a single component service, other users of the service do not need to concern themselves with the underlying functionality or couplings between disparate web services.

To create a combination of web services we first need a graphical representation of the functional components and we decided on gray boxes containing the service name (see Fig. 3. This representation resembles the way the Hive system [14] represents its agents.

To connect two functional components X and Y the user holds down “Shift” and drags the mouse from one box to another. This implies that the output of the first functional object should become one of the inputs of the other. But the second component may have several input fields which
prompts the connection dialog to be displayed (Fig. 4).

The connection dialog asks the user if he wants to connect the output of X to one of the parameters of Y. The user then selects one parameter that makes sense and the connection is complete. The connection is represented as a graphical arrow (see Fig. 3).

It is the logic capability of the user that dictates if the connection will work or not—the system will not stop a user from making illogical connections. However, it is a simple matter to test connections to make sure they work properly, and consequently, in particularly difficult cases, trial and error can always be used.

When a connection is made it affects the graphical components (that were created to interact with the service) accordingly:

Assume that X, Y and Z are functional components that require parameters \((X^1, X^2, \ldots X^n)\), \((Y^1, Y^2, \ldots Y^n)\) and \((Z^1, Z^2, \ldots Z^n)\).

- If X is connected to the parameter \(Y^1\) of Y, the graphical component that represents \(Y^1\) will disappear because it becomes superfluous (the user does not have to write in the value. It is taken from the output of X).
- If X is connected to Y, the button that invokes X will disappear. This happens because the button of Y will invoke X when it is pressed (chain reaction).

We also need some logic to handle situations where connections are not allowed. This logic should function as follows:

- If there is a connection between X and Y, i.e. the X output is connected to a parameter of Y, the user should not be able to make a connection the other way around i.e. from Y to X (direct feedback), because this would lead to an infinite loop.
- If X is connected to Y and Y is connected to Z neither Z nor Y can be connected to X (indirect feedback in the Z case and direct feedback in the Y case). This would lead to an infinite loop.
- Only output from one functional component can be connected to a parameter. When a connection is made the parameter is considered occupied.

Trees of connected functional components are executed recursively starting from the functional component that still has an activation button (see Fig. 5).
2, 3, 5, 1, 4, 6

If you create a connection between 2 and 4 the execution order changes like this (Fig. 6):

1, 2, 4, 2, 3, 5, 6

Or

2, 3, 5, 1, 2, 4, 6

Figure 6. A new connection has been created between 2 and 4.

Notice that functional component 2 has been executed twice in both sequences. This means that more executions than necessary have been made, e.g. every time a “tree” executes every functional component only needs to execute once.

Introducing a session-state, which holds information about the functional component (if it has been executed or not), solved this problem. The execution order changed to be the same as when we did not have a connection between 2 and 4 (see Fig. 5).

3.2.1 Built-In Functions

When you create services you sometimes need to add very simple functions such as basic arithmetic or logic. In those cases it is better to use built-in functions. For example, it is not necessarily efficient to make a network call to add two numbers, it can be done on the client side.

Here are some of the built-in functions we have implemented:

- A repeater. With a repeater you can schedule repeated invocations of a web service or a execution tree. For example, it is possible to schedule stock quotes to be sent by Short Message Service (SMS) every hour
- Arithmetic. We include the four basic arithmetic operators

4. Sharing Services

Given the creative freedom that the sView architecture provides, it is conceivable that the body of future sView users will generate a large number of different services. With the ServiceDesigner it is easy to share these services among users: the creator of a service can publish the service to a web site for others to download, or he can simply send the service to another user by electronic mail or other electronic means. In the Future Work section below we will touch upon another such possibility.

However, the ServiceDesigner has a built-in feature that directly enables a user to become an individual service provider. Each service that is generated using ServiceDesigner is given its own SOAP enabled interface. After the service has been generated and it is loaded into sView, it can expose a new SOAP interface that corresponds to whatever parameters and function the newly created service exhibits. This feature enables the sView briefcase to act as a web service end-point exactly in the same manner as other web service engines.

5. Related Work

The concept of combining pieces of information or functional components into a new composite service may be found in several other systems of which we will mention two: Data Tiles and InfoBeans.

5.1. Data Tiles

The DataTiles system [16] is a platform where end-users place plastic tiles on a big touch-sensitive screen. As soon as a tile is laid on the screen it becomes interactive and it is possible to lay several tiles adjacent to each other to form combinations.

DataTiles is a modular system in which the designers have tried to integrate physical and graphical interaction. To accomplish this, the designers divided the system into three parts:

- Transparent plastic tiles that each has a unique ID
- A large touch sensitive screen, which identifies the plastic tiles when they are laid on the screen
- A computer connected to the system

When a tile is placed on the screen it is identified and a program (i.e. service) that has previously been associated with the tile starts in the computer. The program creates a graphical interface exactly under the tile and the area of the tile is lit up. Users can then interact with the service by touching the tile with a special pen.

The idea behind DataTiles is that the tiles should function as small entities, which can be used separately or be
The user is encouraged to create logical chains, just like you create complex sentences by using small words.

DataTiles is similar to ServiceDesigner in that end-users are able to combine tiles (or functional components as we call it) into new types of services with a common graphical interface.

However, the difference is that we dynamically build the graphical user interface for each functional component (which corresponds to a tile), which means that we can use services that were not intended (or known ahead of time) for the ServiceDesigner. A DataTile (used like a service) has to be tailor-made for the system—if you were to place a tile with an unknown ID on the screen, the system would not know what to do. And, in addition, all required combinations of tiles must be pre-defined. Since each tile (or rather the service represented by the tile) has a known interface that describes its functionality, to connect to it, another tile must use this interface.

On the other hand, ServiceDesigner services are less interactive than their DataTile counterparts. Our services are more akin to “pressing the submit button” as this is the only possible event. This does, however, result in a system that is more open and flexible to new functionality, and in services that are appropriately interactive for a mobile or ubiquitous setting.

5.2. InfoBeans

InfoBeans [5] lets end-users configure their own individualized information services from different web sites.

An InfoBean is a container that holds a small part of an existing web page. The user selects which part of the page the InfoBean should hold. The system then trains itself with the help of the user to understand how to parse out the right information even though the page has changed i.e. to learn the structure of the web page and therefore be able to understand how to handle the category of web pages that it represents.

By collecting several infoBeans in an infoBox (a DHTML based web page) the information from several web pages can be gathered. Every infoBean has input and output channels, which makes it possible to connect infoBeans to transfer data from one infoBean to another.

The infoBean system is conceptually similar to ServiceDesigner together with Sview in that both systems use independent services that can be combined in one common graphical user interface. But there are a few things that differ.

The most fundamental difference is the choice of functionality. InfoBeans uses heuristic methods in order to find the wanted content in the web page; ServiceDesigner uses strictly defined and well described web-based functionality\(^5\), which leads to a more reliable system. We do not have a “server-side” that needs to process heuristic methods leading to scalability problems if you cannot distribute the processes on the “client-side”.

Second, we dynamically build a graphical user interface to the parameters of the service—the InfoBeans system renders HTML in small windows, which leads to a system that bears resemblance to many small and simultaneously open browser windows. We have total control over the actual components and can therefore change and arrange them freely.

Another difference is that the InfoBeans system lacks an overlying framework like sView. When using sView we can have other services, not just simple information services, open at the same time.

6. Future Work

Thanks to the current capabilities of ServiceDesigner it will hopefully be easy for end-users to create and exchange services with each other. Furthermore, it is obvious, based on the success of today’s popular peer-to-peer file sharing applications, that sView users would benefit from a network in which they could more conveniently share their services with each other. We envision using the built in peer-to-peer support (BriefcaseConnectivity, as outlined in section 1.1) of sView to bring complete service availability to the sView platform.

There are two ostensible advantages in creating a network of sView briefcases:

- Users will have continuous and ubiquitous access to new information and services
- It will be possible to design services that leverage the community of sView briefcases including, but not limited to:
  - Trust chains built on the linking of trust from user to user within the sView community (c.f. [1])
  - Rating services that track the performance and quality of services
  - Social mechanisms for recommending services
  - Alert mechanisms that build on the experiences of members of the community to protect the community as a whole from malevolent services

A phenomenon common to many peer-to-peer networks is described by what economists call the network effect [10, 12]. The network effect states that as the network increases in size, the value of a network to an individual also

\(^5\)The format or syntax, but not necessarily the semantics, are well-defined
increases. That is, as more and more resources enter the network, the more valuable the network is to the individual. We hope to generate our own network effect by enabling sView users to participate in a network of sView briefcases, thereby stimulating service creation and use. The purpose of this work is to create a network comprised of sView briefcases that supports the sharing of services and which protects individual users by a number of trust mechanisms.

When the number of services increases, there is a risk that it will be harder to ensure their quality and the user’s security. And with increasing numbers of individually created services, we can no longer depend on the good standing and reputation of the service provider when deciding to use or not use specific services. We therefore believe it is of great importance to provide a platform such as sView/ServiceDesigner with tools and support for trust of services. In a separate project which is scheduled to commence late 2002 we will design and implement such support.

7. Summary and Conclusions

We wish to leverage the use of web services to empower users to become individual service providers, and to demonstrate our ideas we have developed an sView service called ServiceDesigner. It assists the user in the following steps: First, it lets a user access web service descriptions. This entails getting information on what kind of functionality the service provides and then being able to choose which functionality to use. Second, it automatically generates a default graphical user interface for the chosen functionality. The user can visually modify the graphical user interface to his liking. Finally, ServiceDesigner takes the web service description and the designed user interface and generates the necessary programming code to make the new service sView compatible. When the integration is finished we have an sView service that can be loaded into the personal service briefcase or distributed among other users. The finished service also exports its own service description, as a web service, thus making the user’s sView briefcase act as a web service provider.

By enabling users to produce sView-compatible services with ServiceDesigner, we increase end-users’ freedom of combining, individualizing, and personalizing web service based functionality. We believe that the web service architecture is the web-developer model of the future, where professional service providers develop small pieces of functionality that end-users, with the assistance of different tools—like ServiceDesigner—can combine into personalized services. More services implies more benefit to users and with the positive feedback of a growing service base we may well see end-users becoming individual service providers.

8. Acknowledgments

This work was partially financed through the sView SITI^6 project [18]. The authors wish to thank the rest of the OASIS team at SICS, including Markus Bylund, Anna Sandin, Lucas Hinz, Stina Nylander, and Andreas Espinoza.

References


---

\(^6^\)Swedish Information Technology Institute


