

# Brassau: Automatic Generation of Graphical User Interfaces for Virtual Assistants

Michael Fischer   Giovanni Campagna   Silei Xu   Monica S. Lam  
Computer Science Department  
Stanford University  
{mfischer, gcampagn, silei, lam}@cs.stanford.edu

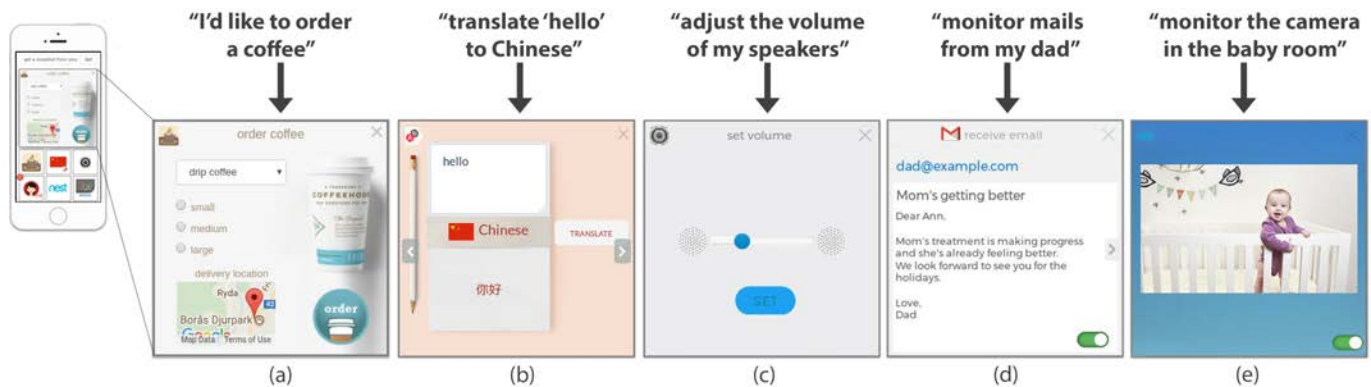


Figure 1: Brassau automatically generates interactive graphical user interfaces for virtual assistant commands in natural language.

## ABSTRACT

This paper presents Brassau, a graphical virtual assistant that converts natural language commands into GUIs. A virtual assistant with a GUI has the following benefits compared to text or speech based virtual assistants: users can monitor multiple queries simultaneously, it is easy to re-run complex commands, and user can adjust settings using multiple modes of interaction. Brassau introduces a novel template-based approach that leverages a large corpus of images to make GUIs visually diverse and interesting. Brassau matches a command from the user to an image to create a GUI. This approach decouples the commands from GUIs and allows for reuse of GUIs across multiple commands. In our evaluation, users prefer the widgets produced by Brassau over plain GUIs.

## ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces

## Author Keywords

Graphical user interfaces; Aesthetics; Visual preferences; Virtual assistants.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

MobileHCI '18 September 3–6, 2018, Barcelona, Spain

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5898-9/18/09.

DOI: <https://doi.org/10.1145/3229434.3229481>

## INTRODUCTION

This paper presents a graphical virtual assistant called Brassau<sup>1</sup> that automatically generates an interactive GUI from users' natural-language commands.

## Design of Brassau

The interface of our Brassau prototype, on an Android phone, is shown in Figure 1. Users can type in their command in a text box at the top of the screen, or say the command by using Android's voice input support. Brassau automatically translates each command into a widget. Up to seven widgets can be shown on one screen: one main widget and six minimized at the bottom. Users can interact with the main widget directly. If a widget is updated while it is minimized, a notification badge appears; tapping a minimized widget will turn it into the main widget, and the state from its last interaction will be shown. Users can manage their widgets by reordering them with a press-and-hold interaction on the icon, or delete them by tapping the "x" on the upper right corner of the icon. They can also toggle on and off monitoring functions with a switch, as shown in Figure 1(d,e).

GUIs have some significant benefits over text and voice-based interfaces used in traditional virtual assistants. With Brassau's widget-based GUI interface, users can keep track of their *personally favorite* virtual assistant commands *at a glance*, which

<sup>1</sup>Brassau is a chimpanzee whose paintings were mistaken as painted by a human artist [30].

makes monitoring multiple commands possible. Users can issue *one-click commands*, which are particularly useful for commands that are reused often and that have many parameters. Additional benefits, discussed below, correspond to the examples (a)-(e) in Figure 1.

- (a) With a GUI, users can see *what parameters are needed to fulfill a command* and easily specify them. For example, when ordering coffee, user can quickly change the type, size, and delivery location.
- (b) Users can *reuse widgets* without having to respecify every aspect of them. Once the translation widget has been set up, users can type in the text they want translated without specifying the input and output languages.
- (c) GUIs can provide *interactive controls* to users. For example, moving a slider to adjust controls, such as volume, is more intuitive than saying, “Brassau, turn down the volume” repeatedly.
- (d) With notification badges on minimized widgets, users can *monitor selected updates*, such as emails from specific senders, and maximize the one of interest. Additionally, for privacy and efficiency reasons, it is preferable to *skim long sequences* of results, such as email messages, rather than have them read aloud.
- (e) Users can see *live graphical updates* in the main widget, such as a baby monitor.

This paper presents a novel algorithm that automatically generates GUIs from virtual assistant commands. Our focus is not just creating functional widgets, but widgets that are aesthetically appealing. To do so, we propose a top-down approach that starts with a conceptual design rather than a more conventional bottom-up constraint-based technique.

We created a knowledge base of GUI templates that are well-designed layouts of input and output parameters on visually interesting backgrounds. Templates can be created in a few minutes. Our algorithm finds the template that best matches the inputs and outputs of a given virtual assistant program.

## Contributions

The contributions of this paper include the following:

- A novel GUI-based virtual assistant that supports glanceable notifications, repeating and reusing tasks, and controlling background tasks with the help of a graphical interface.
- An algorithm that generates interactive GUIs automatically from natural-language commands. The algorithm generates diverse, interesting, and semantically relevant GUIs by matching the commands to a template in the knowledge base and filling the template. The growth of virtual assistant commands can be handled by adding templates.
- User studies show that users prefer GUIs stylized by Brassau over plain GUIs, GUIs semantically match the users’ commands, templates can be reused, and templates can be easily created.

## RELATED WORK

Automatically generating graphical user interfaces is an interdisciplinary endeavor. Our work draws on prior research in the areas of multimodal user interface systems, automatic generation of GUIs, computationally aiding humans in the exploration of design spaces, and computationally understanding visual design.

### Multimodal Virtual Assistant

The majority of virtual assistants nowadays are conversational agents with a chat-based interface. Research has been done on multimodal interfaces to enable a richer and more natural user experience [13, 20, 23]. Although these works give virtual assistants a multimodal interface, the GUIs are hard coded to specific functionality. With the rapidly increasing abilities of virtual assistants, the solution will be difficult to scale. In contrast, in this paper we present a scalable algorithm that allows for the reuse of GUI templates across commands.

Amazon’s Alexa [1] added a graphical user interface by introducing the Echo Show. Echo Show uses a built-in touch display to show information and do basic interactions. Six fixed plain templates are provided for developers to choose from and developers have to manually specify what content to show and where the content should be placed in the templates. Existing skills cannot be supported by the Echo Show without manual developer work. In contrast, we automatically generate GUIs given a command referring to skills in a virtual assistant repository, and new skills can be supported without the need for developer intervention.

### Automatic Generation of GUIs

Research systems have been developed to automatically or semi-automatically lay out user interfaces. These approaches use optimization-based techniques to arrange graphical components such as buttons and menus using different constraint-based metrics [6, 8, 19, 22, 27, 31].

SUPPLE takes constraint-based methods a step further, by not only generating the layout but also selecting the appropriate interactors and dividing the interface into navigational components [9, 10]. However, these works do not take into account higher-level concepts such as visual design and aesthetics. Our solution takes into account both high-level visual concepts as well as constraints specified by the program.

### Computer Augmented Exploration of Design Space

Work on having computers aid humans while designing interfaces has been a rich area of research. Computational tools can help designers explore a design space and try out new ideas quickly. Sketchplore is a multi-touch sketching tool with a real-time layout optimizer [28]. The tool allows designers to explore the design space quickly and get computational feedback on which designs are promising. The designs produced by the system are “sketch” quality, helpful for prototyping but not as “ready-to-use” GUIs. Webzeitgeist introduces the concept of design mining and provides a tool to help designers parametrically search for visual design examples [15]. ICrafter presents a template-based interactive system which can be dynamically matched to different services [18]. However, creating a template requires coding in a traditional programming

language, which not only makes it harder than our system, but can also cause security issues since it runs arbitrary code. Additionally, ICrafter templates only apply to a specific program signature, unlike Brassau's templates which can be reused for many similar but different programs. Previous works are targeted at developers, not end users.

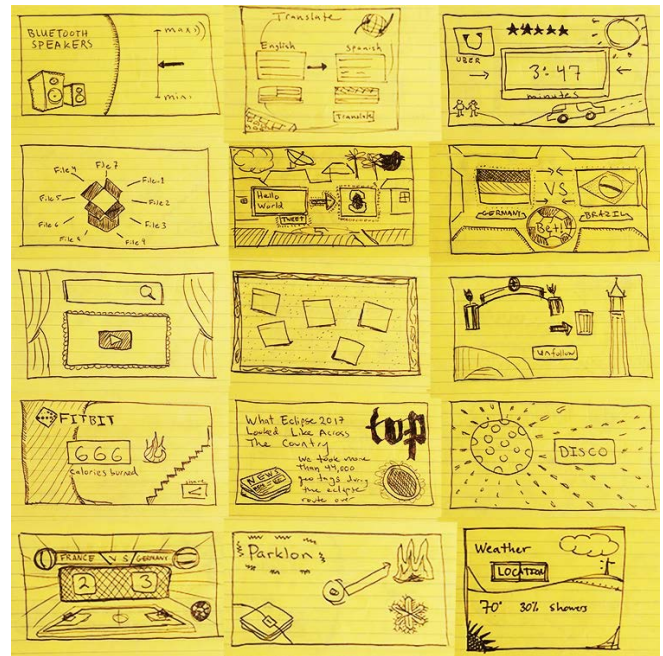
### Computational Visual Design and User Experience

More complicated than the strict problem of finding an optimal layout is finding an optimal visual design. The design landscape for visual design is larger and more user specific. To make a system that can automatically generate user interfaces, we must define what makes an interface good. Prior work has recognized the importance of visual appeal in an application and developed ways to measure it. Lindgaard et al. show that the visual appeal dominates the first-impression judgments of perceived usability and trustworthiness [16]. The study by Bateman et al. compares the interpretability and recall of plain and visually embellished charts [3]. They find that visual embellished charts are recalled notably more without impact to interpretability. These studies support the need for having computationally generated interfaces that are functional and aesthetic, which is a contribution of this paper. Although good design has significant impact, it is time consuming to build. Programmatic generation of appealing designs has been studied for typesetting [14], tag clouds [2, 12], data visualization [21], and natural-language queries [11, 24, 25]. These techniques are domain specific for relatively similar elements and do not generalize to more complex visual optimizations.

### ALGORITHM DESIGN PROCESS

We used an iterative design process to derive the solution proposed in this paper to automatically build GUIs. To start, we compiled a list of the most popular virtual assistant commands using logs from the Almond open-source virtual assistant project [5]. In addition to exploring how people currently use virtual assistants, we conducted interviews to explore what people wished their virtual assistant could do, but was not capable of yet. An author of the paper sketched out what an ideal GUI would look like for top commands, if it was laid out by a designer. The results, shown in Figure 2, highlighted the need of a good layout for elements corresponding to the inputs and outputs and a semantically relevant and interesting background to make the interface visually attractive.

In our first iteration of a solution, we built a bottom-up assembler that would take in a program specification and assemble the components using a generative model. The model analyzed the program and mapped the input and output parameters to graphical elements in a repository, such as icons, buttons, colors, and backgrounds. For interactive controls, we relied on the standard browser controls, combined with the Bootstrap library of components [17]. We set the background color to a gradient of colors extracted from the logo. However, we found that when the individual elements were put together, the elements did not match with each other and looked visually inconsistent. The UI elements had different styles and varying levels of quality. We curated the quality of the UI elements but the resulting designs still looked inconsistent because of variations in style. Overall, the designs looked formulaic and



**Figure 2:** After we found the most commonly used commands issued to virtual assistants, we sketched out what an ideal GUI would look like for each. The designs acted as a point of reference for comparison with our results.

not like the sketches made by hand. This experience taught us that more entropy is needed in the design process to create a visually interesting set of widgets.

In response, instead of using a bottom-up approach, we investigated using a top-down, knowledge-intensive approach. Can we leverage the large corpus of existing images and apps to create diverse, well-designed, and semantically relevant GUIs?

We experimented with ways in which we could match existing GUI templates to commands automatically. To do so, existing GUIs needed to be augmented with additional semantic information. So, we created a template markup language called BANANA<sup>2</sup> and an interactive tool so users can provide the necessary information quickly.


### A FUNCTIONAL BRASSAU

Now that we have described the top-down approach we take in designing the system, we next describe the system architecture to make Brassau functional. Brassau is based on *Almond*, a text-based virtual assistant that translates natural-language commands into executable programs and executes them [5]. Such programs use APIs that are stored in Thingpedia and are written in a programming language called ThingTalk. Brassau uses Almond to translate natural language into ThingTalk programs and to execute ThingTalk programs. Instead of Almond's textual output, Brassau automatically creates GUIs for ThingTalk programs so users can interactively change input parameters and see the results.

<sup>2</sup>Brassau's Adaptive Non-specific Annotations for Novel Assistants

Command Type	Example
WHEN	monitor Apple stock
WHEN with predicates	monitor when bob@example.com emails me
GET	search tweets about “Pierre Brassau”
GET with predicates	list files larger than 10 MB in my Dropbox
DO	tweet ‘hello world’

**Table 1: Primitive Thingpedia command types with example sentences.** Thingpedia is a database of APIs that Brassau uses to generate functional GUIs.

<b>Name</b>	@com.twitter.search
<b>Inputs</b>	req query : String opt count : Number
<b>Outputs</b>	text : String hashtags : Array(Hashtag) urls : Array(URL) from : Username in_reply_to : Username
<b>Short Description</b>	search
<b>Long Description</b>	search \$query on Twitter
<b>Icon</b>	
<b>Multiple Results?</b>	yes

**Table 2: The Thingpedia entry for @com.twitter.search.**

### Thingpedia: A Database of APIs

Thingpedia is a crowdsourced and open-source virtual assistant skill repository. There are three kinds of primitive skills: WHEN commands monitor events, GET commands retrieve data, and DO commands perform an action or display data to a user. Each command calls a function in Thingpedia, along with optional predicates such as equality or string containment to filter the results. Example WHEN, GET, and DO commands are shown in Table 1.

Each Thingpedia entry has an API declaration that includes a list of input parameters, some of which are optional, and a list of output parameters. An example Thingpedia entry for Twitter is shown in Table 2. Each parameter has a name and a type. A Thingpedia entry also contains a short and a long description of how the API can be referred to in natural language. The former shows how a user can issue a command using the API; the latter is used to produce a canonical sentence fully capturing the semantics of the command that can be shown to the user to resolve any ambiguity or defects in parsing. Each entry also includes an icon for the service, and an indication of whether it returns multiple results.

### ThingTalk: Virtual-Assistant Commands

A ThingTalk program is a single WHEN-GET-DO statement, which combines up to three primitive commands from Thingpedia. For example, the following command monitors the current weather, and when the temperature is below a threshold, generates and posts on Twitter a picture:

```
monitor @weather.get_current(location = here),
temperature ≤ 5C
⇒ @meme.generate(template = “brace yourselves”)
⇒ @twitter.post_picture(url = meme.url,
caption = “winter is coming”)
```

At runtime, if a program requires an authenticated API, the system asks the user for a username and password if they have not been entered already. The program is then executed and passes the results to the chat interface to present to the user.

### Almond: Converting Natural Language into ThingTalk

Brassau accepts textual virtual-assistant commands, either entered by the user directly, or translated from users’ voice commands using a third-party speech-to-text software. Brassau uses Almond to translate the text to ThingTalk. Almond uses a deep-learning neural network based on a sequence-to-sequence model [26] to translate a natural-language sentence into a ThingTalk program.

### Brassau Widget Generation

Brassau takes a ThingTalk program generated by Almond and creates a GUI for it. It looks up the appropriate Thingpedia entries to interpret the input and output parameters in the program. If the user modifies parameters in the GUI, the ThingTalk program is updated. Whenever the user runs the program, it is passed to the Almond runtime and results from Almond are passed back to Brassau to display.

There are three kinds of Brassau widgets:

- WHEN widgets are activated when a specified trigger occurs. Users can turn on or off the widget with a switch. If the widget is turned on, the program is sent to the Almond runtime, which returns a unique identifier for the running instance. Brassau keeps an open connection to Almond where it receives new output values as the program executes. When the widget is turned off, Brassau also informs Almond correspondingly. If the WHEN widget is minimized, an incrementing notification badge will appear to notify the user when the rule has triggered. Example WHEN widgets are shown in Figure 1(d,e).
- GET widgets are run immediately once they are created. Subsequently, the user can change the input parameters and run the program again by pressing the refresh button. Example GET widgets are shown in Figure 1(b).
- DO widgets are activated when the user presses the *action* button. The text in the *action* button is the first word of the short description of the last primitive in the program. Example DO widgets are shown in Figure 1(a, c).

Every widget has a logo, which is generated from a combination of logos in the Thingpedia entries and input parameters in the program. Every widget also has a title, formed by combining the short descriptions of all primitives in the program; it is shown when maximized.

### Input Parameters

A ThingTalk program, translated from natural language, is often underspecified due to missing input parameters. Therefore, the GUI needs extra input parameters to run the program. Brassau maps the input parameter types in Thingpedia into input elements, according to the mapping shown in Table 3. For numeric types, a slider is used if the range is known or can be inferred from the unit of measure. For enumerated



ThingTalk type	Input element
String short	Text entry
String long	Text area
Number, Measure	Slider or text entry
Bool, Enum(on, off)	Switch
Other enum	Radio or drop down
Location	Map
Entity	Logo and name if specified, else drop down
Picture	File picker
Contact	Picture and name
Color	Color picker
Other	Constant value if specified, else text entry

**Table 3: The mapping between input ThingTalk types and input elements. Entity is the type of well-known public objects, such as sport teams or companies.**

types, a radio button is used if there are three options or fewer, otherwise a drop-down menu is used to save space in the widget. As a special case, if the enumerated choices are *on* and *off*, a switch is shown. For hashtags and usernames, a # or @ symbol is shown before the input. Entity types, such as company names and languages, are shown with an icon and a name from Thingpedia when specified. See, for example, the language entity “Chinese” in Figure 1(b). If the language is not specified, the user is shown a drop-down menu with all the language options.

### Output Parameters

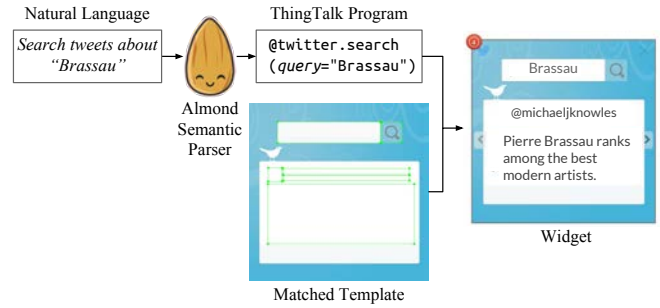
Brassau maps output parameter types in Thingpedia into display elements, as shown in Table 4. For example, parameters of type URL are turned into clickable links, and parameters of type Measure have their units shown. For ThingTalk functions that return multiple results, each result is mapped to a different page of the widget. The user can switch pages with left and right arrows, and the badge at the top indicates the number of unread results.

ThingTalk type	Display element
Number, Measure	Value + Unit
Contact	Link to contact (tel: or mailto:)
Picture	Image
URL	Clickable link
YouTube video ID	Embedded video
Other	Text

**Table 4: The mapping between output parameter types in ThingTalk and output user-interface elements.**

### USER-INTERFACE TEMPLATES

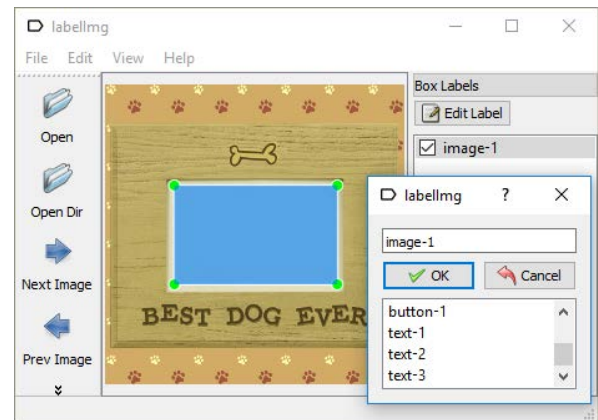
So far we have described how Brassau accepts a natural language input, uses Almond to translate it into a ThingTalk program, and maps parameters to control and display elements. Next Brassau matches the ThingTalk program with a *template* to generate a widget, as shown in Figure 3. Here, we introduce templates as semantically tagged images with replaceable areas for control and display elements. We have developed TemplateMaker, an interactive template creation tool, and BANANA, a template markup language.



**Figure 3: Generating a widget with Brassau. The user input is translated by Almond into a ThingTalk program, which is combined with a matched template to create a widget.**

### Template Creation

Template are created using TemplateMaker, an application we developed by customizing LabelImg [29] with a script that generates BANANA templates. Images are first pre-processed by cropping them to the correct aspect ratio, removing extraneous elements, and adding new elements if desired. The image is then loaded into TemplateMaker, where we draw semantically tagged overlaying rectangles on UI elements, as shown in Figure 4. Once the image has been tagged, it is run through a post-processing script that marshals the tagging data and combines it with semantic data and a dominant color palette extracted using the Color Thief library [7] to produce a BANANA template.



**Figure 4: The interface of TemplateMaker, used by Brassau template creators to define rectangular overlays and tag them by their types. Once created, a template can be used by any program.**

### The BANANA Template Markup Language

The BANANA markup language is used to annotate GUIs so that they can be matched with generated programs. The formal definition of BANANA is provided in Table 5. The primary component of a BANANA template is the background image, which can either be a direct screenshot or a modified image. The user provides a list of semantic tags that describe the contents of the background image.

Template $t$	: $img\ tag^*\ box^*\ cc\ pal$
Image $img$	: $\langle picture \rangle$
Tag $tag$	: $[\text{required} \mid \text{optional}] \langle word \rangle$
Box $box$	: $P_{top-left} P_{bottom-right} c_{dom} c_{top} c_{left} c_{bottom} c_{right}$ $type\ h\ align?\ fsz?\ ffam?\ cover?$
Point $p$	: $\langle x \rangle \langle y \rangle$
Color $c$	: $\langle r \rangle \langle g \rangle \langle b \rangle$
Type $type$	: $input \mid text \mid image \mid title \mid map$ $\mid slider \mid button \mid switch \mid logo$
Hierarchy $h$	: $\langle number \rangle$
Alignment $align$	: $left \mid right \mid center \mid justify$
Font Size $fsz$	: $\langle number \rangle$
Font Family $ffam$	: $serif \mid sans \mid monospace \mid handwritten \mid display$
Corner Colors $cc$	: $c_{top-left} c_{top-right} c_{bottom-left} c_{bottom-right}$
Palette $pal$	: $c^4$

**Table 5: The formal definition of BANANA, a markup language that captures the layout, semantics, and colors of a GUI template.**

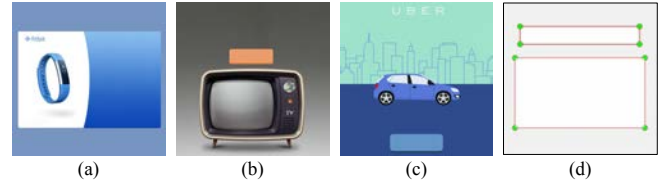
Users draw boxes over the image, each of which has a type attribute, such as *image*, *text*, or *button*, which determines how it is supposed to be used. Each box also includes a rank ordering; larger and more prominent boxes should be ranked first. A box can also be annotated with text alignment, font size and font family, from a set of 5 predefined families. These annotations exist to give a template designer more control over the output of Brassau.

While the boxes are tagged manually, other parts of the BANANA template are generated automatically. The color palette for the whole image is extracted, using the *modified mean quantization algorithm* [4], which clusters colors according to their distance in the RGB space. The extracted color palette is used to match the API's logo to a template. For example, if Slack is used in a ThingTalk program, a template with Slack colors should be used. The template's color palette is also used to choose the colors of UI elements, such that they can be made to match the color scheme of the image. Associated with each box is its dominant color, which is used to choose a contrasting color for text elements to increase readability.

The system can handle variability after it has chosen a template. The matching algorithm may choose a template with more boxes than needed, in which case it is going to have to fill unused boxes. Additionally, there are elements, such as logos, that a user might want to selectively cover. To fix these problems, we introduce the *cover* attribute; its presence indicates that the box should be covered. The colors on the sides of the box are used to construct a gradient that covers the box and blends in with the rest of the image.

### Template Collection

To understand what makes a good template, we put together *PrimitiveSet*, which consists of all the working and distinctly different functions from the 203 primitives in Thingpedia. *PrimitiveSet* has altogether 91 functions. We built a database of templates for the set by looking for relevant images on the Internet. Note that copyright of these images must be obtained if they are used commercially. In the process, we identified four useful categories of templates: *screenshot-based*, *skeuo-*



**Figure 5: Examples of template designs: (a) screenshot-based, (b) skeuomorphic, (c) semantic, and (d) neutral.**

*morphic*, *semantic*, and *neutral* templates. Examples of each are shown in Figure 5.

### Screenshot-Based Designs

As widgets are specialized versions of existing apps, it is natural and appealing for them to adopt the style of the original, professionally designed apps. Such templates can be found from screenshots of the official app or website. Contributors to Thingpedia may wish to supply such templates as they create entries for their devices or services. Annotating these types of templates is straightforward because the images have clearly delimited interactive areas and could be done automatically in the future.

### Skeuomorphic Designs

High-quality skeuomorphic templates can be derived from real-world images of physical objects like scoreboards, clocks, and TV remote controls. These templates are useful for displaying results. We found that the skeuomorphic designs should be front facing images of objects, avoid distracting details, and have enough empty space to display the content.

### Semantic Designs

Semantic designs contain graphic elements that share semantics with the programs, such as specific images or brand decorations. These have the advantage that they are applicable to widgets of different services with similar semantics.

### Neutral Designs

It may be the case that none of the templates in the knowledge base match a program. We create a set of neutral templates for the most common program signatures, as a catch-all to handle APIs that have no matching templates. Neutral templates have a flat color background, manually placed controls, and no decorations.

## TEMPLATE MATCHER

To match a program with the best template in the collection, we calculate a heuristically defined cost function for each template and choose the one with the minimum cost. The cost function has three components:

- Semantic,  $S$ , quantifies how the template tags in the templates match the words in the program description.
- Hue,  $H$ , measures if the background color matches that of the program logo. We do not want to recolor the template based on the logo to avoid creating unnaturally colored templates.
- Layout,  $L$ , measures the layout fit between parameters and boxes. Details of this function are described below.

Box Type	Input/Output	Element Type
Slider	Input	Slider
Switch	Input	Switch
Map	Input	Map
Input	Input	Text entry
		Text area
		Radio
		Drop down menu
		File picker
Image	Output	Picture
		Video
Text	Output	Text output
		Entity logo and name
		Contact picture and name
		Link

**Table 6: Brassau’s algorithm assigns control/display elements of a widget to boxes in a template according to the mappings in this table.**

Let  $C(t, p)$  be the cost function for a program  $p$  using template  $t$  in collection  $T$ , and  $\hat{t}$  be the best template for  $p$ ,

$$C(t, p) = \alpha S(t, p) + \beta H(t, p) + \gamma L(t, p)$$

$$\hat{t}(p) = \arg \min_{t \in T} C(t, p)$$

where  $\alpha$ ,  $\beta$  and  $\gamma$  are heuristically assigned coefficients.

There is some flexibility as to how many inputs and outputs a widget needs to show. Twitter, for example, has many optional input parameters for search. Our algorithm tries to show as many inputs and outputs that can fit into a template based on priority. The highest priority is given to unspecified but required input parameters, as a program cannot run otherwise. Output parameters are next in priority; after all, that is the purpose of the widget. Priority is given to picture and video output types since that is why people want a graphical virtual assistant. Interactive elements, such as switches and sliders, have higher priority than input texts as they often serve the primary function of the widgets.

In order of priority, Brassau’s algorithm assigns each input or output element to a box of matching type, according to Table 6. If there is no matching box, it tries to assign the parameter to a box of a different type. A cost is assessed for every mismatched assignment, and for every unassigned element and unassigned box.

### STYLIZING THE APPLICATION

The final step of creating the widget is to bring all the elements together by stylizing them. To color the input and output elements, we calculate colors that are legible and look good within the context of the overall design. We determine them programmatically by calculating the contrast ratio between each of the palette colors and the color of the box the element will be filling. To calculate the contrast, we compute the ratio between the luminance of the two colors.

The palette color with the highest contrast is chosen to be the foreground color for the box. This makes the text match the other decorative elements in the template, as in Figure 1. If none of the colors have a contrast ratio greater than 2.5, a generally accepted usability guideline to ensure readability,

we choose the color using a *split-complementary color scheme*. This scheme is known to have enough contrast.

To display as much of the text as is reasonable, we adjust the font size dynamically based on the content, with minimum and maximum sizes, to best fit within the box. This displays shorter information, such as stock prices or step counts, more prominently.

### EVALUATION

We created two fully working prototypes of Brassau, a web version and an Android version. We evaluate ideas presented in this paper and our prototypes by asking the following questions:

1. How diverse are the templates?
2. Can templates be reused?
3. Can templates be generated easily?
4. Does Brassau generate visually appealing widgets?
5. Do the widgets match the intent of the program?
6. How do users respond to the end-to-end experience?

#### How Diverse are the Templates?

To get a sense of how diverse templates can be, we studied the template knowledge base we created for PrimitiveSet introduced earlier. We first note that the 91 commands in PrimitiveSet are diverse themselves: 54 of the commands use private data and the rest use public resources. 23 are WHEN, 34 are GET, and 34 are DO commands.

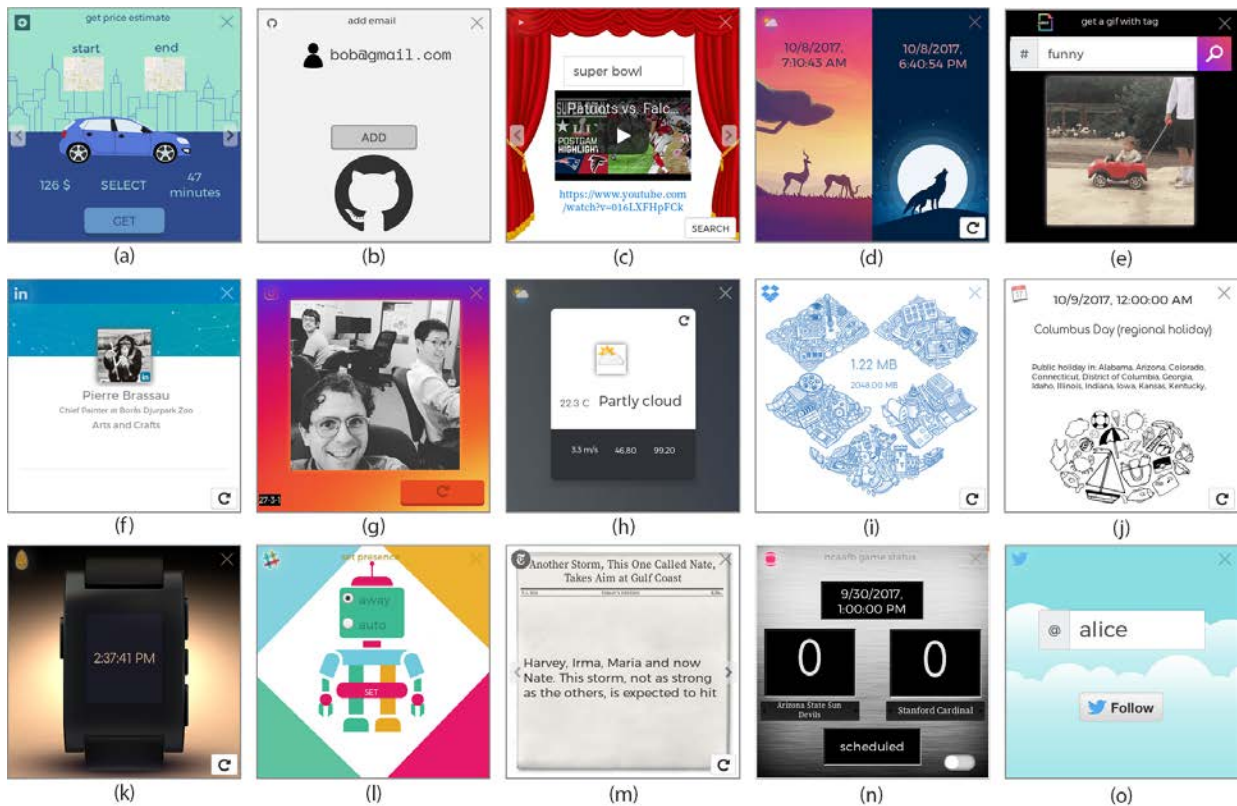
We collected 65 templates for PrimitiveSet. There are 20 screenshots, 11 skeuomorphic, 25 semantic, and 9 neutral designs. 27 templates have a strong brand association, such as the use of brand colors or additional logos. 29 have semantic decorations. 5 templates have no branding differentiation, used for services like weather or SMS, making them reusable for other purposes.

Of the 25 semantic templates, 3 include a real picture of the device they represent, and the rest use drawings or flat colors. Of the other 31 non-neutral templates, 2 are monochrome or line-art, 20 use flat colors or simple gradients, and 9 use realistic colors. 25 of the templates include at least one input box, 34 at least one text box, and 14 at least one image box. 6 templates are annotated with an explicit position to place the logo, which otherwise would be placed at the upper left position. 34 templates have a box for the action button to execute the program.

Brassau generated fully functional widgets for all the commands in PrimitiveSet, a sampling of which are shown in Figure 6. The results demonstrate how diverse the generated GUIs are, in terms of look-and-feel, colors, layouts, and styles. 41 of the generated widgets are interactive, showing one or more input parameters that can be changed, and 7 of which include either a switch or a slider. Brassau hides one or more inputs for 15 of the widgets. 16 widgets show a list for results.

#### Can Templates be Reused on Compound Tasks?

Our premise is that the template-based algorithm can handle new commands, without necessarily requiring new templates.



**Figure 6:** The results of Brassau when given the following natural-language commands: (a) how much will Uber cost from Stanford to San Francisco? (b) add bob@gmail.com to my Github account, (c) search ‘super bowl’ on Youtube, (d) what time is the sunrise and the sunset? (e) show me gifs with hashtag #funny, (f) get my Linkedin profile, (g) show me an image from my Instagram feed, (h) what is the weather today? (i) get my Dropbox quota, (j) when is the next holiday? (k) what time is it? (l) set my status on Slack, (m) show me the news from New York Times, (n) monitor the scores of the Stanford Cardinals football team, and (o) follow @alice on twitter.

#	Command
(a)	Search YouTube and play it on TV.
(b)	Generate an Archer meme and send it to Slack.
(c)	Translate Washington Post headlines in Lifestyle section to Italian.
(d)	Tweet the latest NASA Astronomy Picture of the Day.
(e)	When the security camera detects something, call 911.
(f)	Send a GIF to Gmail.
(g)	Tweet my Instagram pictures.
(h)	Set my phone to vibrate every day at 9 am.
(i)	When the Apple stock price is below \$100, order me a large coffee.
(j)	Auto reply to my emails.
(k)	Send my weight to doctor@example.com.
(l)	Tweet the score when a Warriors game ends.

**Table 7:** Compound commands in CompoundSet.

To test this premise, we created *CompoundSet*, a set of 12 compound commands that combine two primitives, as shown in Table 7. The signatures for these commands, the input and output types, are a hybrid of primitives, which are distinct from those in *PrimitiveSet*.

Brassau generated functional widgets for these compound commands, without requiring any new templates to be added. The results of the first four commands in Table 7 are shown in Figure 7. Parameters from the two primitives are combined

using a semantically relevant template. Brassau correctly generated widgets for all the 12 cases. A neutral template was used in one case, because it was the only one matching the signature. For two cases, Brassau showed only the output parameters for the DO primitive and no input parameters, for a lack of a better template choice.

Despite the relatively small number of available templates, Brassau created working widgets for new compound commands, demonstrating the effectiveness of the ranking and matching heuristics in Brassau’s algorithm.

### Can Templates be Created Easily?

To test the ease of template creation, we recruited 5 students to add a template of their choice to the database. Two of the students had previously taken an HCI course and the other three had not. Users were asked to redesign their least favorite widget in *PrimitiveSet*. One user designed a new interface for playing YouTube videos, one for setting the temperature on the thermostat, and three for a news widget. To make a new template, they looked for an appropriate image from the web and used our TemplateMaker to tag it. All users found the process and software easy to use and completed the task in less than ten minutes. One user commented, “All I did was make three boxes and rank the order and it was done. It knows what



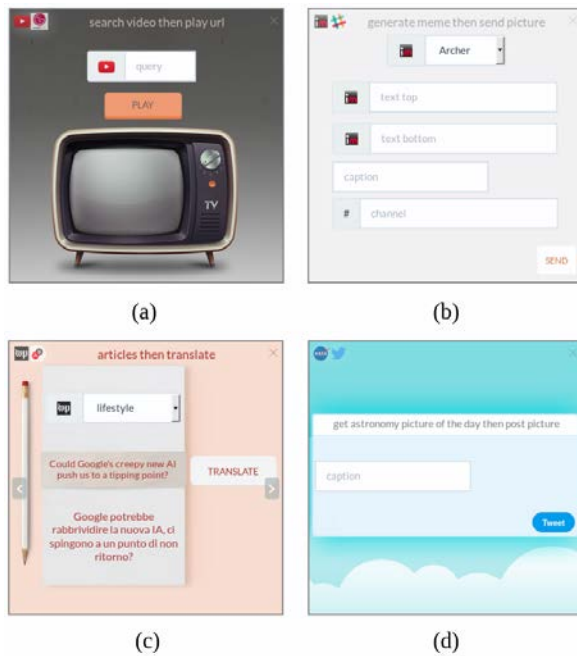


Figure 7: Automatically generated widgets for compound commands (a), (b), (c) and (d) in Table 7.

is the most important and does it.” All users were satisfied with the final result and said that they would use the tool to build interfaces for personal use.

### Can Brassau Generate Visually Appealing Widgets?

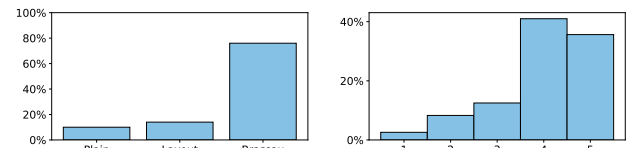
In this experiment, we conducted a study to evaluate which template layout type do users prefer. We experimented with three graphical designs as follows:

- *Plain*. The input and output boxes selected to be presented by Brassau are laid out, centered, sequentially from top to bottom, and with no background.
- *Layout*. The inputs and output are placed into the boxes of the best-fit template. However, the elements are not styled and the background is not shown.
- *Brassau*. The full algorithm is used, which includes the background and the custom layout.

To find which graphical designs users liked the most, we hired 40 Amazon Mechanical Turk workers, located in the United States, who had an approval rating above 95%. For each app, we created three presentations using the three designs described above and showed them to the worker in random order. The worker was asked to choose the one that they would most likely use, along with a brief explanation describing their decision.

Overall, the widgets with the Plain, Layout, and Brassau design were picked 10%, 14%, and 76%, respectively, as shown in Figure 8(a). We observed that the users fall into two groups: (1) 10% of the users prefer simpler widgets, and choose the Plain and Layout designs more than 70% of the time. (2) 90% of the users prefer the Brassau widgets, choosing them more

than 70% of the time. This means that Brassau is preferred by most people. Furthermore, this highly skewed distribution of preferences suggests we should let users configure globally how they want their widgets presented. In this way, we can satisfy all users.



(a) Percentage of votes for each design method. (b) Distribution of ratings on the match between commands and GUIs.

Figure 8: Evaluation of Brassau widgets.

### Do the Widgets Match the Intent of the Program?

We recruited 100 Mechanical Turk workers to help evaluate if the generated GUIs match the intent of PrimitiveSet. Workers are shown pairs of a natural-language command and the generated widget. They are asked to rate how well the GUI matches the intent of the command using a 5-point Likert scale, from strongly disagree (1) to strongly agree (5). The average of all the ratings is 4.0, and 77% of the ratings are at level 4 or 5, as shown in Figure 8(b). This suggests that Brassau widgets match the commands well.

The 5 highest ranked pairs have more variety in their colors; they are either screenshots, or semantic templates that contain an image of the object. None of the top 10 commands use a neutral template. Widgets with less colors tend to have lower ratings. Four of the five lowest pairs are black and white or neutral; the fifth has a solid yellow background. Commands that contain uncommon words, such as “xkcd” and “meme”, tend to have lower ratings, perhaps because workers are unfamiliar with such commands and would not know what their GUIs should look like.

GUIs that show a semantic link between the command and the template have the least variance in ratings received. For widgets that show a picture of a word in the command, such as a dog, thermostat, or TV, the standard deviation is between 0.6 and 0.7. However, the five highest standard deviations, 1.2-1.4, are found among the lowest quartile of rated widgets, suggesting that there is a greater difference of opinion at the low end.

### End-To-End Experience

Our Brassau prototype, built on top of the experimental Almond infrastructure, is not mature enough to support a user study in the wild. Nonetheless, we wanted to get some feedback to inform our next steps. We conducted an end-to-end pilot study with the same five users of the template creation study. We presented users with a cheat sheet showing 350 WHEN, GET, and DO example commands. We asked them to enter commands of their choice into the text box of Brassau’s UI on an Android phone, as shown in Figure 1. If the users struggled with entering the commands in natural language,

the authors intervened and suggested a different command or a paraphrase of the same command. Evaluations lasted approximately 30 minutes.

Users felt that the commands they entered were well represented by the GUIs. P1, P3, and P5 commented that they liked the simplicity of the GUIs. As long as the natural-language input was understood by Almond, users did not experience usability issues while interacting with the system. One user said, “it’s clean, easy to use.”

When shown the list of 91 widgets from PrimitiveSet, different users found different apps useful. Examples of widgets they liked include the music player, translation, memes, and the light bulb. While one liked the Uber widget, another did not because they would rather use the original app. One liked the slider to set a volume because they thought it would be easier to use than controlling a system with voice, one did not. The difference in opinion of what is good demonstrated the importance of users being able to personalize their dashboard.

User liked the idea of a *graphical* virtual assistant. Two of the users commented on the overall usefulness of having a dashboard of customized GUI widgets for their favorite devices or services on their mobile phone. P1 liked the system and described Brassau as a “phantasmagoria of functions.” They liked to see the news and chat with friends in the same app. P2 commented that the system was like Facebook because they could do many tasks within the same app. They would like to piece together a custom app for all their favorite functions.

The users gave some concrete suggestions on the visual presentation. P3 wanted the news widget to contain an image as well as text and P2 wanted it to use the real New York Times font. They observed that the most visually appealing widgets had strong decoration and short pieces of content. They specifically liked the Uber widget, which shows a car stylized with the Uber colors. None of the users commented on the stylistic inconsistency across widgets.

All users commented that they wanted the minimized widgets to be smaller, so that the main widget would have more room to display information. P3, P4, and P5 commented that the list display would be better if they could swipe instead of clicking left and right.

Four of the users, P1, P3, P4, and P5 wanted to do more with the content once it was displayed. P1 tried a compound command that would upload newspaper articles to Facebook, and was confused that they did not have the opportunity to change what would be posted. They wanted additional actions such as being able to share the images and seeing other related content. Confirmation, sharing, and other general functionalities can be added into the Brassau framework in the future and made available to all the widgets.

## LIMITATIONS AND FUTURE WORK

Brassau’s algorithm currently can only represent one ThingTalk program in one widget. We found in our end-to-end user studies that users want more complex widgets that allow them additional features such as sharing and linking to other widgets. Future work will consider showing contextual actions

in a widget and how users can combine multiple ThingTalk programs to make widgets more interactive.

As we evaluated Brassau’s limitation, we also identified ways in which the Thingpedia type system could be improved. We recommend adding a concept of records to Thingpedia, to make relationships of data explicit. For example, in a sporting event, we need records to associate the winning score with the winner team, and the losing score with the losing team.

In this implementation, each widget has its own style. If users want a consistent style across widgets, we envision two possible solutions. First, as the database of templates grows, the templates can be organized into families of templates that are consistent with each other. This is analogous to the notion of themes for app icons. Second, we can add a final module that restylizes all the widgets to be consistent, possibly using deep-learning techniques.

We hypothesize that it is possible to generate some basic templates automatically using deep-learning techniques. In our current system, we manually created simple, neutral templates to be used if there were no other matching templates. Once the template database is populated with enough training examples, a deep-learning algorithm could be developed to generate neutral templates automatically.

## CONCLUSION

This paper introduces Brassau, a graphical virtual assistant that automatically generates interactive widgets from natural-language commands, translated by Almond to ThingTalk. Brassau allows users to assemble together a dashboard of personalized widgets, making it easier to repeat commands and see multiple results at a time.

Brassau introduces a novel template-based method that leverages a large corpus of GUI templates to create visually diverse and interesting GUIs. Users create a GUI template by taking a background image and tagging the bounding boxes for control and display elements, ranking them, and adding semantic tags. The template also contains color information automatically extracted from the image. Brassau matches the ThingTalk program to a template and stylizes it to create the GUI. Brassau is the first template system which matches its program dynamically based on types and semantics.

Our experiments show that Brassau’s algorithm is general and supports the diverse space of virtual assistant commands. In our user studies, we found that user prefer widgets built using Brassau to widgets that did not have the Brassau background or UI layout. End-to-end user tests suggest that the system is useful, easy to understand and easy to extend.

## ACKNOWLEDGMENTS

Support for this work was provided in part by the Stanford MobiSocial Laboratory, sponsored by AVG, Google, HTC, Hitachi, ING Direct, Nokia, Samsung, and Sony Ericsson. The authors also thank Bastian Pfleging and the anonymous reviewers for their helpful comments.

## REFERENCES

1. Amazon Alexa 2017. (2017).  
<https://developer.amazon.com/alexa>
2. Scott Bateman, Carl Gutwin, and Miguel Nacenta. 2008. Seeing Things in the Clouds: The Effect of Visual Features on Tag Cloud Selections. In *Proceedings of the Nineteenth ACM Conference on Hypertext and Hypermedia (HT '08)*. ACM, New York, NY, USA, 193–202. DOI: <http://dx.doi.org/10.1145/1379092.1379130>
3. Scott Bateman, Regan L. Mandryk, Carl Gutwin, Aaron Genest, David McDine, and Christopher Brooks. 2010. Useful Junk?: The Effects of Visual Embellishment on Comprehension and Memorability of Charts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2573–2582. DOI: <http://dx.doi.org/10.1145/1753326.1753716>
4. Dan Bloomberg. 2008. Color quantization using modified median cut. Leptonica.  
<http://www.leptonica.com/papers/mediancut.pdf>
5. Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. ACM Press, New York, New York, USA, 341–350. DOI: <http://dx.doi.org/10.1145/3038912.3052562>
6. Dennis JMJ De Baar, James D Foley, and Kevin E Mullet. 1992. Coupling Application Design and User Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*. ACM, 259–266. DOI: <http://dx.doi.org/10.1145/142750.142806>
7. Lokesh Dhakar. 2017. Color Thief. (2017).  
<http://lokeshdhakar.com/projects/color-thief/>
8. James Fogarty and Scott E Hudson. 2003. GADGET: A Toolkit for Optimization-Based Approaches to Interface and Display Generation. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. ACM, 125–134. DOI: <http://dx.doi.org/10.1145/964696.964710>
9. Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: Automatically Generating User Interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI '04)*. ACM, New York, NY, USA, 93–100. DOI: <http://dx.doi.org/10.1145/964442.964461>
10. Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically Generating Personalized User Interfaces with Supple. *Artificial Intelligence* 174, 12-13 (Aug. 2010), 910–950. DOI: <http://dx.doi.org/10.1016/j.artint.2010.05.005>
11. Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G. Karahalios. 2015. DataTone: Managing Ambiguity in Natural Language Interfaces for Data Visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software Technology (UIST '15)*. ACM, New York, NY, USA, 489–500. DOI: <http://dx.doi.org/10.1145/2807442.2807478>
12. Yusef Hassan-Montero and Victor Herrero-Solana. 2006. Improving Tag-Clouds as Visual Information Retrieval Interfaces. In *International Conference on Multidisciplinary Information Sciences and Technologies*. 25–28.
13. Michael Johnston, John Chen, Patrick Ehlen, Hyuckchul Jung, Jay Lieske, Aarthi M Reddy, Ethan Selfridge, Svetlana Stoyanchev, Brant Vasilieff, and Jay G Wilpon. 2014. MVA: The Multimodal Virtual Assistant. In *Proceedings of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL '14)*. 257–259. DOI: <http://dx.doi.org/10.3115/v1/W14-4335>
14. Donald E. Knuth and Michael F. Plass. 1981. Breaking paragraphs into lines. *Software: Practice and Experience* 11, 11 (1981), 1119–1184. DOI: <http://dx.doi.org/10.1002/spe.4380111102>
15. Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R Klemmer, and Jerry O Talton. 2013. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3083–3092. DOI: <http://dx.doi.org/10.1145/2470654.2466420>
16. Gitte Lindgaard, Cathy Dudek, Devjani Sen, Livia Sumegi, and Patrick Noonan. 2011. An Exploration of Relations Between Visual Appeal, Trustworthiness and Perceived Usability of Homepages. *ACM Transactions on Computer-Human Interaction* 18, 1 (2011), 1:1–1:30. DOI: <http://dx.doi.org/10.1145/1959022.1959023>
17. Mark Otto, Jacob Thornton, Chris Rebert, Julian Thilo, and others. 2011. Twitter Bootstrap. (2011).  
<http://getbootstrap.com/>
18. Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. 2001. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *International Conference on Ubiquitous Computing*. 56–75. DOI: [http://dx.doi.org/10.1007/3-540-45427-6\\_7](http://dx.doi.org/10.1007/3-540-45427-6_7)
19. Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, and Jean Vanderdonckt. 2016. A Layout Inference Algorithm for Graphical User Interfaces. *Information and Software Technology* 70 (2016), 155–175. DOI: <http://dx.doi.org/10.1016/j.infsof.2015.10.005>
20. David Reitter, Erin Marie Panttaja, and Fred Cummins. 2004. UI on the Fly: Generating a Multimodal User Interface. In *Proceedings of HLT-NAACL 2004: Short Papers (HLT-NAACL-Short '04)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 45–48.  
<http://dl.acm.org/citation.cfm?id=1613984.1613996>

21. Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. DOI: <http://dx.doi.org/10.1109/TVCG.2016.2599030>
22. Andrew Sears. 1993. Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout. *IEEE Transactions on Software Engineering* 19, 7 (1993), 707–719. DOI: <http://dx.doi.org/10.1109/32.238571>
23. Ethan Selfridge and Michael Johnston. 2015. Interact: Tightly-coupling Multimodal Dialog with an Interactive Virtual Assistant. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction (ICMI '15)*. ACM, New York, NY, USA, 381–382. DOI: <http://dx.doi.org/10.1145/2818346.2823301>
24. Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. 2016. Eviza: A Natural Language Interface for Visual Analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 365–377. DOI: <http://dx.doi.org/10.1145/2984511.2984588>
25. Yiwen Sun, Jason Leigh, Andrew Johnson, and Sangyoon Lee. 2010. Articulate: A Semi-automated Model for Translating Natural Language Queries into Meaningful Visualizations. In *International Symposium on Smart Graphics*. Springer Berlin Heidelberg, 184–195. DOI: [http://dx.doi.org/10.1007/978-3-642-13544-6\\_18](http://dx.doi.org/10.1007/978-3-642-13544-6_18)
26. Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112. <http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>
27. Pedro Szekely. 1990. Template-Based Mapping of Application Data Interactive Displays. In *Proceedings of the 3rd annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UST '90)*. ACM, 1–9. DOI: <http://dx.doi.org/10.1145/97924.97925>
28. Kashyap Todi, Daryl Weir, and Antti Oulasvirta. 2016. Sketchplore: Sketch and Explore with a Layout Optimiser. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*. ACM, 543–555.
29. Tzutalin. 2015. LabelImg. Git code. (2015). <https://github.com/tzutalin/labelImg>
30. Wikipedia contributors. 2018. Pierre Brassau — Wikipedia, The Free Encyclopedia. (2018). [https://en.wikipedia.org/w/index.php?title=Pierre\\_Brassau](https://en.wikipedia.org/w/index.php?title=Pierre_Brassau)
31. Brad Vander Zanden and Brad A Myers. 1990. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*. ACM, 27–34. DOI: <http://dx.doi.org/10.1145/97243.97248>