Reducing implementation time for the GUI design-to-code process using DSL

Charilaos Mulder

September 6, 2016

Supervisor(s): Lynda Hardman

Signed: Signees
Abstract

To implement a GUI, a software engineer needs more information than a designer’s original design document offers. Therefore, designers create supporting documentation such as wireframes, explicit specifications and interactive prototypes in order to better convey their intention. Software engineers then need to interpret these design deliverables in order to correctly implement the GUI. While this GUI design-to-code process works, it consumes considerable amounts of time and potentially results in a loss of accuracy with respect to the original design. Here we propose an automated workflow, where the original design is first converted to code through an intermediary language called DSL. This language supports a broad range of GUIs, and forms the basis for automated translation to working code. Combining existing research and our own analysis of a set of GUIs we have derived the requirements for DSL to be able to describe most GUIs and thereby sufficiently support our proposed GUI design-to-code workflow. With the requirements of DSL specified, this paper is a first step in the realisation of our goal to reduce the implementation time for GUIs while at the same time increasing the accuracy of the end result.
1 Introduction 7
2 State-of-the-art in the GUI design-to-code workflow 9
3 Critical analysis of the GUI design-to-code workflow 13
4 Proposed GUI design-to-code workflow 17
5 The Design Specification Language 21
  5.1 Requirements for DSL 21
  5.2 Comparison to the state-of-the-art in UI markup languages 27
  5.3 Examples of DSL specification 29
6 Conclusions 31
7 Future work 33
8 Appendix 37
CHAPTER 1

Introduction

Almost all software actively used by consumers and professionals has a graphical user interface (GUI). Therefore, any software engineer that takes his/her user base seriously makes sure their software has a well designed GUI. In the context of GUIs, ‘well designed’ means satisfying the following aspects, distilled from the often cited design principles by Dieter Rams, as applied to software design[1, 2].

Functional design provides the necessary functionality for the workflow and, ultimately, the desired purpose. In addition to the goals a user needs to accomplish with an interface, the workflow should be effective and not require the user to perform redundant operations.

Usable design makes sure an interface is accessible by its intended users. Understandable navigation structure, appropriate information hierarchy, readable text and discoverability of functions are examples of qualities that lead to usable design[3].

Aesthetic design is what makes an application pleasant to look at and use. Delightful visuals and animations can turn dull software into a more joyful experience. Functionality and usability should be the starting points for GUI design, rather than aesthetics. Moreover, aesthetics should not interfere with usability or else the cognitive effort and usage time will be affected.

Often, a software engineer needs to team up with a GUI designer for their expertise to end up with software that satisfies these requirements. After the designer has finished the GUI design, it needs to be handed over to the software engineer in a way that allows it to be accurately reproduced in code. The transferred design consists of many parts, including, but not limited to, mockups of application screens, individual design assets such as button shapes, interactive prototypes demonstrating the workflow and navigation, and a written specification clarifying attributes such as spacings and font styles. The software engineer then has to interpret these design deliverables. This GUI design-to-code process currently requires a considerable amount of time and effort to retain the intentions of the graphic designer.

As Calvary and Coutaz put it [4]:

The development of user interfaces (UIs), ranging from early requirements to software obsolescence, has become a time-consuming and costly process. Typically, the graphical user interface (GUI) of an interactive system represents about 48% of the source code, requires about 45% of the development time and 50% of the implementation time, and covers 37% of the maintenance time (Myers and Rosson,1992) [5]. These figures, evaluated in the early nineties, are increasing dramatically with the spread of new interaction techniques such as vocal and gestural modalities, resulting in additional requirements (Petrasch, 2007)[6].

The intended end users of this research are those who see potential in the idea of leveraging a designer’s visual and behavioural specification to reduce a software engineer’s implementation
time of GUIs. This is achieved through more directly transferring design information to the
software engineer. An important contribution to this transfer is the breakdown of GUI elements
to their most fundamental properties, which are the building blocks of fully functional interfaces.
In addition to reducing GUI overhead time in the workflow for software engineers, this research
is also of interest to those who want to understand the fundamental assembly and behaviour of
GUI elements.

In this paper, we first outline and analyse the traditional way in which software engineers receive
GUI designs for implementation. We then propose an approach for a GUI design-to-code process
that aims to make GUI implementation for software engineers faster through automation and
more precise by retaining more information of the original design. This automated workflow
includes a GUI specification language which we name DSL (Design Specification Language).
Thereafter, we analyse and structure the requirements for DSL.
In order to implement a GUI, a software engineer needs to receive a design with instructions on what to implement. While this step is inevitable, the manner in which this happens can differ in practice. To understand how a design is received by a software engineer, it is important to know of which parts a design consists.

Functional, usable and aesthetic design were mentioned in the introduction as the qualities required for a well-designed GUI. These qualities should be reflected in the end product. However, these are not the parts a designer transfers to a software engineer to work with. Designers break up their design in the following deliverables which are transferred as implementation instructions [7]. Note that this list includes only deliverables transferred to software engineers. The reason is that the design of a GUI, often referred to as user interface (UI) design is usually part of a broader user experience (UX) design, not all of which is directly relevant to implementation.

**Figure 2.1: Illustration of a wireframe**

**Wireframes** are low quality static representations of the GUI. They state what content (such as images and text) and interface elements go where. Wireframes can be annotated with brief specifications about the layout or interactions [7]. Details are omitted to save time and cost, thus wireframes alone are insufficient for a software engineer to be fully instructed on how to implement the GUI.
Mockups are high quality static representations of the GUI. The software engineer should be able to get a clear understanding of how the GUI looks without animations or interactions. When mockups are available, wireframes are of little use to software engineers. Mockups are usually direct image exports of the original design document.

Assets are individual items such as button shapes or background patterns that are directly used in the implementation. These are usually delivered in the form of raster/vector images, but can also include text or video and they are sometimes accompanied by code snippets for interaction behaviour.

Specification is written to define all requirements for the GUI that cannot be derived or precisely measured from other design deliverables. A specification can include size, position, typeface information, visual information and precise interaction/animation/navigation logic.
The design document itself is sometimes handed to the software engineer. The philosophy behind this method is that the software engineer can independently retrieve visual information, such as sizes, spacings, colors and typefaces. Using the source design document eliminates the need for the designer to write some of the specification and reduces back-and-forthing with the designer about unspecified visual information.

Interactive prototypes provide the software engineer with high quality interactive mockups on the device the GUI is intended for. In this way, a software engineer can navigate through the GUI, understanding the navigational structure, the interaction model as well as animations. Designers can easily make interactive prototypes using tools that let them interconnect mockups by interactions and assign animations.
CHAPTER 3

Critical analysis of the GUI design-to-code workflow

In this chapter, we evaluate the transfer of design deliverables to software engineers, described in chapter 2. This evaluation focuses on shortcomings surrounding the transfer of deliverables, motivating a new approach for the GUI design-to-code workflow, to be described in chapter 4. The analysis presented here is effectively fully our own. Although the odd reference exists that discusses problems with GUI specifications (see e.g. [8]), we were not able to identify any pertinent literature.

Figure 3.1: Design deliverables covering properties required by software engineers

In essence, the desired end result of the transfer is that all elements of the GUI design end up in the implementation: rapidly and correctly. Figure 3.1 illustrates a single GUI element with a few of its properties required for implementation, as well as how design deliverables each cover part of these properties. The figure serves an explanatory purpose and does not incorporate the exhaustive list of possible GUI element properties or design deliverables. With this figure and the deliverables from chapter 2 in mind, we can identify the following problems.

Deliverables can overlap in the properties they cover. A common example is that mockups and specification documents can both hold information about the color of elements and relative
spacing between them. Valuable time is consumed by the designer defining the same property multiple times. In addition, overlap provides no added value for the end result. Instead, it increases the opportunity for contradicting information, resulting in a loss of time if detected or a loss of accuracy if overlooked. The potential of overlapping specifications to cause inconsistencies in designs has been identified for development of OO-software [9], but, as far as we are aware, not in the context of GUI specifications.

**Deliverables can have unintended gaps** in the properties needed for implementation when design deliverables do not cover all of them. Gaps are missing information to software engineers, requiring them to lose time by communicating with designers to retrieve the missing information. Worse, in case the software engineer freely interprets the design because of an information gap, the resulting GUI suffers a loss in accuracy by deviating from the original design.

**Deliverables can be incorrect** as a result of the designer making reading, memorising or typing mistakes when analysing the original design document to create the deliverables. Incorrect instructions to the software engineer will likely end up as inaccuracies in the GUI implementation since they are harder to detect than instructions that are missing.

In addition, overlapping deliverables are extra work compared to the original design document, as illustrated in figure 3.2. The designer puts effort making these deliverables for the software engineer to more easily decipher the design specifics. The need for this transfer of information comes from the fact that software engineers are not the same persons as the GUI designers and therefore require design properties in specialised formats for best cognition.

Figure 3.2: Overlapping design deliverables are extra work for the designer

![Diagram](image)

**Figure 3.3: Implementing visual information and interaction logic that is already present**
Last but not least, when analysing the designer’s original design document and deliverables it becomes clear **most visual information and interaction logic is already present** and, in a way, gets re-implemented by software engineers. As an example, with design documents holding the precise visual properties, put into interactive prototype tools, valuable information about shapes, fonts, colors, interactive objects and their clickable area, animation, scrolling areas and more stands by, ready for quick and accurate automation.
In this chapter we connect the dots of the critical analysis in chapter 3 to introduce a new solution to the GUI design-to-code process. The ins and outs of this new approach are explained and the way a software engineer adopts it is clarified.

Software has a number of aspects where software engineers need to find solutions to and write code for, as illustrated in figure 4.1. The GUI is often one of the larger aspects a software engineer has to deal with, in part because of the need to interpret and implement a GUI designer's design deliverables. The workflow could use improvements in speed and accuracy, while reducing complexity in the transfer of design to the software engineer.
Our proposed approach to solving the problems in the GUI design-to-code workflow is as follows. As illustrated in figure 4.2, a GUI starts as a design document and ultimately ends up in code. A design document is the first and most detailed digital manifestation of a designer’s ideas. Code is the digital solution of a software engineer following the requirements that needed implementation. In our proposed approach, the only intermediate document is a Design Specification Language document, from now on referred to as a DSL document. DSL herein has the role of a language that is used to specify the entirety of a GUI. DSL documents consist of a list of GUI elements, each defined by its properties. The list of properties should be exhaustive in visual and behavioural possibilities and therefore sufficient to define any GUI element and its role in the GUI. DSL fits in a workflow in which design resources can be converted into DSL documents, which in turn can be converted into GUI code to be used by the software engineer, also illustrated in figure 4.2. Furthermore, DSL imposes a number of additional requirements, described below.

**Figure 4.2: Proposed GUI design-to-code workflow through an intermediary language**

**Figure 4.3: DSL allows for modular input and output**

**DSL is intended for modular output.** Defining GUI elements by their fundamental properties makes it more manageable to convert it to multiple output platforms, taking into account platform differences such as human input methods or screen sizes. Converters to specific platforms can interpret element characteristics so as to convert them into the closest corresponding native elements of the output platform. In its current state, the scope of DSL is limited to platforms with mice and trackpads pointers, keyboards and/or on-screen (multi)touch as input methods.

**DSL is intended for modular input.** In order not to limit the possibilities of the GUI
designer, s/he should be able to choose his/her own tools to design the GUI with. Be it Sketch, Illustrator or less popular design applications - even more mathematical approaches which design information architecture and interaction flow rather than aesthetics - should be eligible as input for conversion to DSL. A prerequisite in this case is that the design application has an open document format that can be read by a converter.

**DSL is intended for automation.** Each GUI element in DSL is defined by a list of properties which are mandated to have values. Any values lacking in the source design document can be filled with best-case defaults. This is not necessarily practical when trying to write or read a DSL document by hand, but it is beneficial for converters having perfect knowledge of all GUI element properties. In this way, converters know what to expect and deal with those properties and defaults in their own way. It allows for converters to platforms not yet thought of at present.

The next chapter we describe a first step in the process of realising our proposed approach.
CHAPTER 5

The Design Specification Language

In this chapter, we state the requirements for DSL derived from our own research on functional GUIs. We then compare the capabilities of the hypothetical language DSL, defined by its requirements, to the state-of-the-art in other languages which could be used to fit in our proposed workflow. The chapter ends with an impression of what GUI elements in DSL could look like.

5.1 Requirements for DSL

When trying to distill the essence of a GUI, one might think of elements such as buttons, sliders, text or logos. However, to impose no limits to the creative possibilities of a design as an input for DSL and, in order to preserve all possible functionality in a software GUI that is generated from DSL, a "button" is too specific an element for total freedom. For DSL not to be restrictive, it needs to consist of smaller building blocks from which GUI elements such as buttons are made.

Before diving into the exact requirements for DSL, an example of well known GUI elements follows to show how three GUI elements differ from each other in their fundamental properties, as well as how these elements hold similarities that are less apparent on first sight.

Radio buttons. These are a combination of components: usually text preceded by bullets, where the selected bullet is the active choice. The multiple radio button entries are usually listed vertically. Other arrangements are possible, too. To change the active entry, the user clicks any other entry. The bullets are a clickable area for pointers - and in better implementations - the
Tabs. When compared to radio buttons, the layout in space can be different: tabs are mostly horizontal. They can also be visually different: the active tab indication is usually done through boldness, difference in color, an underline or a combination of these. Another visual difference is that often icons are used or a combination of icons and text, while radio buttons tend to only have text. However, the interaction for changing tabs is identical and so is the output logic that dictates only one tab can be active.

Repeat mode button in iTunes. This is a button that changes between modes when clicked (play song list once, repeat song list, repeat current song). Each mode is represented by its own icon. The interaction model is different compared to radio buttons and tabs: keep clicking the icon until the desired repeat mode comes along. Visually, it differs because it only shows one mode at a time, making it more compact. Logically, the output is still the same as radio buttons and tabs: only one option is active at a time.

In the aforementioned examples, we can identify a number of GUI element properties that are needed to visually and functionally represent three different GUI elements that accomplish the same task (choosing one of many options).

For DSL to function in the workflow mentioned in chapter 4, it needs to be based on the fundamental properties of GUI elements. To break down the aspects of GUI elements to their basic properties, we researched a variety of existing GUI applications on desktop, mobile and web, which we list in the Appendix. Barfield [11] also provided valuable knowledge about fundamental interactions and on the way content can be displayed on screen. The Amsterdam Hypermedia Model [12] further refined our knowledge of navigational structure and the relation between spacial layout and time.
The search for all fundamental properties resulted in, what we hope to be, an exhaustive list of possible properties of which GUI elements consist, including their descriptions and value descriptions for clarification. The resulting list of properties is presented in Table 5.1 below.
<table>
<thead>
<tr>
<th>Element property</th>
<th>Property description</th>
<th>Value description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (ID)</td>
<td>Unique identifier of element</td>
<td>Element name</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Identifier of a globally defined element type. Inherits visual and behavioural properties.</td>
<td>Type name</td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Horizontal size of an element. Absolute size or relative size compared to screen width or another element.</td>
<td>Pixels or percentage of screen width or other element</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Height</td>
<td>Vertical size of an element. Absolute size or relative size compared to screen height or another element.</td>
<td>Pixels or percentage of screen height or other element</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Horizontal position</td>
<td>Horizontal position of an element. Absolute or relative distance from left side / horizontal centre / right side of screen or another element.</td>
<td>Pixels or percentage, from screen position or other element</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Vertical position</td>
<td>Vertical position of an element. Absolute or relative distance from top / vertical centre / bottom of screen or another element.</td>
<td>Pixels or percentage, from screen position or other element</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Depth position</td>
<td>Relative layering position of an element.</td>
<td>Element name of layer underneath</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Depth attachment</td>
<td>Defines whether or not the element is attached to the element underneath. If attached, it moves along with the element underneath.</td>
<td>Boolean</td>
<td>Appendix</td>
</tr>
<tr>
<td>Content</td>
<td>Any asset file containing image, video, plain text, markup text or other common file types such as PDF.</td>
<td>Asset file name and extension</td>
<td>Appendix</td>
</tr>
<tr>
<td>Content layout</td>
<td>Defines the way the content of an element is displayed within the size of an element.</td>
<td>Values such as &quot;fit&quot;, &quot;fill&quot; or &quot;stretch&quot;</td>
<td>Appendix</td>
</tr>
<tr>
<td>Direct visual styling</td>
<td>Visual styling information about the element. Only displayed if no asset file is present.</td>
<td>List of CSS styling properties</td>
<td></td>
</tr>
<tr>
<td>Event trigger</td>
<td>Actions that can trigger an Event. This property defines whether or not an element is interactive and/or dynamic.</td>
<td>None, one or more of the following types: manual (human interaction), automatic (directly) or co-event (Event listens to Event trigger of other element)</td>
<td>Appendix</td>
</tr>
<tr>
<td>Element property</td>
<td>Property description</td>
<td>Value description</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
<td>-------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Trigger delay</td>
<td>Delay time for Event after the occurrence of an Event trigger.</td>
<td>Milliseconds</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Manual interaction</td>
<td>Defines the types of human interaction methods, if present by &quot;manual&quot; Event trigger.</td>
<td>None, one or more of the following types: hover, click/tap, long-click/tap, release of click/tap, pressure sensitive click/tap, drag, right-click, scroll (direction), defined coordinates and directions for custom gestures, display rotation or a keyboard key (combination)</td>
<td>Appendix</td>
</tr>
<tr>
<td>State</td>
<td>Defines whether or not the element is active at present. Applies to interactive elements only. Examples of inactive elements are Undo buttons when no undo's are available, or a button made inactive by a pop-up on top.</td>
<td>Boolean</td>
<td>Appendix</td>
</tr>
<tr>
<td>Interaction logic</td>
<td>Defines the output behaviour of the interactive element.</td>
<td>None or one of the following types: button bell-push, button two-state (switch), radio button, variable/liquid controller (slider: define constant step size or curve, as well as the range), mode controls</td>
<td>[11]</td>
</tr>
<tr>
<td>Interaction transparency</td>
<td>If true, the element area defined by its size passes interaction to the element below. False if the element has Manual interaction.</td>
<td>Boolean</td>
<td>Appendix</td>
</tr>
<tr>
<td>Interaction area</td>
<td>Size of the area that allows for human interaction, likely but not necessarily identical to the size of the element.</td>
<td>Pixels, horizontal and vertical</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Event</td>
<td>Defines the appearance, change of state or disappearance of the element itself and other elements.</td>
<td>Build-in / change of state (change of property values) / build-out, for a list of one or more elements including itself</td>
<td>[12]</td>
</tr>
</tbody>
</table>
With the properties in Table 5.1, functional GUI elements can be described in DSL. However, it is important to note that each property can have multiple values. This is what allows a GUI element to have different animations when appearing or disappearing. Another benefit of multiple values is that an icon can appear differently on specific platforms depending on conventions. As a final example, textual GUI elements should appear differently according to the system language of the device. When multiple values are present, each value can be identified by a tag, such as “appear”, “disappear”, “linux”, “mac”, “english” or “dutch”.

A short example follows to clarify the concept of navigation from one screen within a GUI to another. A navigational element such as a menu button, triggered by manual interaction, uses its Event property to build out the elements on screen and build in the elements of a new screen. In

<table>
<thead>
<tr>
<th>Element property</th>
<th>Property description</th>
<th>Value description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animation</td>
<td>Holds the information about the animation of the element itself and other elements.</td>
<td>End state: change in position/size/rotation/direct visual styling. Animation curve with time in milliseconds, or, tied to drag movement. List of elements involved.</td>
<td>[12], Appendix</td>
</tr>
<tr>
<td>Animation loop</td>
<td>Enables a repetitive animation pattern.</td>
<td>Boolean</td>
<td>Appendix</td>
</tr>
<tr>
<td>Manual moving interaction</td>
<td>Defines the types of movement human interaction can invoke on this element and elements attached to it. While being a form of manual interaction, it works independently.</td>
<td>Static or one or more of the following types: horizontally scrollable, vertically scrollable, zoomable, rotatable</td>
<td>[11], Appendix</td>
</tr>
<tr>
<td>Snap</td>
<td>Snap a scrolling, zooming or rotating view to certain positions/coordinates/degrees.</td>
<td>Boolean and (list of) snap positions/coordinates/degrees for each Moving interaction logic applicable</td>
<td>Appendix</td>
</tr>
<tr>
<td>Stick scroll</td>
<td>Stick scroll position of an element. Primarily used for section headers which stay persistently on screen.</td>
<td>Horizontal and/or vertical position</td>
<td>Appendix</td>
</tr>
<tr>
<td>Scalability logic</td>
<td>Defines screen size ranges in which Events (build-in/change of state/build-out) are triggered. Used to adapt the GUI to different screen sizes by measures such as showing or hiding elements, change spacings by altering positions, replace text by smaller icons and tabs by more compact dropdown menus.</td>
<td>List of screen width and height ranges and corresponding Events</td>
<td>Appendix</td>
</tr>
</tbody>
</table>

Table 5.1: Fundamental GUI elements and their properties
case only part of the screen is subject to change by the navigational element, the Event property can intelligently build out only a subset of the elements on screen [12].

Note that the properties are intended to fulfill the requirements of creating a GUI without reaching into functionality, other than navigating by changing the elements on screen. Applications with custom logic that often require graphics engines such as drawing, 3D modelling or gaming applications are out of the scope of the Design Specification Language.

On an individual level, the listed element properties are comprehensible. However, as a collection of properties that make up a single UI element, there are dependencies between the properties that are best understood when laid out in a dependencies hierarchy, as shown in figure 5.4.

![Diagram of dependencies between fundamental UI Element properties](image)

Figure 5.4: Dependencies between fundamental UI Element properties

The list of properties in table 5.1, including the possibility of multiple values per property, completes the list of required semantics for DSL. We consulted common GUI elements listed by the Nielsen Norman Group [13] and verified that these elements can be built using the fundamental properties derived from our research. Further examples are provided in section 5.3.

5.2 Comparison to the state-of-the-art in UI markup languages

DSL is a vital part to the proposed workflow by intermediating as a language that supports modern GUI requirements. As designing a language from scratch involves a serious amount of work, it is worth considering whether any existing specification language(s), either directly GUI-related or more general, can eliminate the need for DSL to exist. This would only be the case if such an existing language supports all fundamental GUI requirements stated in the previous section, in addition to being convenient for automation with modular design input and modular platform output.

To do so, we discuss a selection of languages based on the list of user interface markup languages on Wikipedia [14], as well as the popular data structuring languages, RDF, XML, SGML and JSON. Other requirements for the candidate languages are a high quantity of citations or other evidence of wide-spread use and, for the UI specification languages, being designed for general purpose GUs instead of specific GUI platforms such as e.g. television screens.
Xcode Interface Builder The Xcode Interface Builder is a full-fledged development package encompassing a very complete range of GUI design elements. However, being both proprietary and targeted exclusively to iOS and macOS, it fails to meet the requirement of platform independence.

MARIA XML The Model-based Language for Interactive Applications (MARIA) describes itself as “… a universal, declarative, multiple abstraction-level, XML-based language for modelling interactive applications in ubiquitous environments.” It is designed to target a wide variety of GUIs using different forms of physical interaction such as touch, vocal or text-based. However, its focus is exclusively on describing the functional aspects of GUIs. It describes its elements as being Interactors or compositions of Interactors. It relies on downstream processing to supply the necessary visual rendering of the elements. As such it decouples the behavioural and visual aspects of a GUI to a degree that imposes obstacles to the unified design process we propose here.

UIML and USIXML Two separate efforts at creating XML-based user interface specification languages are User Interface Markup Language (UIML) and User Interface eXtensible Markup Language (USIXML). The latter, however, appears to have discontinued its development. Of all the languages considered UIML appears to come closest in capabilities to our DSL requirements. It allows both behavioural and visual aspects to be combined in the specification. However, in its current form it is distinctly human-unfriendly like most DTD-based languages, providing obstacles for its use by designers. Moreover, although still alive it appears to have had very limited adoption (a web search for UIML examples yields as a top hit a document dated 1999).

CSS Cascading Style Sheets was developed to render documents using markup from the DTD family of markup languages. Its intent is to separate logical design from visual design. Over the years it has acquired a number of features that allow it to perform GUI related features such as buttons, roll-overs, dropdowns and animations. However, it is strongly tied to (X)HTML and web-based applications. Moreover, it cannot specify a GUI by itself, as it requires underlying native HTML elements to obtain the necessary functionality. For a colloquial critique of CSS as a design language see.

SVG The original design goal of SVG is a system-independent description of scalable vector graphics. While through the addition of additional functionality such as animation, interaction with the DOM and CSS, embedded media, events and scriptability it can be used to describe GUIs of some complexity, it nevertheless is not designed as a generic GUI design framework.

The following four languages are generic mark-up languages for producing both machine- and human-readable structured documents. In principle each of these could serve as the basis for developing a GUI specification language, e.g. XML is at the basis of Maria XML, UIML and USIXML. However, none of them specifically facilitates GUI description, and potentially introduce their own overhead e.g. in file size, with respect to the more focused design goal of DSL.

RDF As its name implies, the Resource Description Framework (RDF) is a framework rather than a single language. Its goal is to facilitate the standardized exchange of information between web-based applications. It allows the definition of elements with a wide variety of attributes. For the definition of the RDF see.

XML The most widely-used of all markup languages, the eXtensible Markup Language (XML) aims at “emphasiz(ing) simplicity, generality, and usability …” Like, RDF it allows the specification of elements with essentially arbitrary attributes. For the definition of XML see.

SGML The Standard Generalized Markup Language (SGML) is the (grand)father of all markup languages. Although in principle the most general of all, it is hardly ever used in its pure form, which is highly verbose. By definition, it possesses all the possibilities of the derived languages such as RDF, XML and HTML. For the definition of SGML see.
**JSON** While intended to be more friendly to human reading than the DTD-based languages, JSON [27] is just a data description format and lacks the language-like features of the former, restricting e.g. its automated searchability [28].

Looking at the GUI requirements comparison, we can conclude that languages other than DSL do not sufficiently fulfill the requirements for an automated workflow that produces a fully functional GUI and/or not currently maintained. Therefore, DSL is needed for our proposed workflow.

### 5.3 Examples of DSL specification

While the DSL syntax is not yet designed, examples can be provided to give an impression of how GUI elements would look in DSL. While ideally the hypothetical syntax should be in line with conventional GUI specification syntax such as CSS to optimize ease of adoption, the next three examples of a plain background, a button and a photo that rotates upon activation of the button omit the implication of a specific syntax. Figure 5.5 illustrates the example GUI elements combined in a vertical display and their DSL values are seen in table 5.2.

![Figure 5.5: Example GUI elements combined](image)

<table>
<thead>
<tr>
<th>Element property</th>
<th>Background</th>
<th>Rotate Button</th>
<th>Photo</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name (ID)</strong></td>
<td>background</td>
<td>rotateButton</td>
<td>photo</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td>width of screen</td>
<td>64px</td>
<td>width of screen</td>
</tr>
<tr>
<td><strong>Height</strong></td>
<td>height of screen</td>
<td>64px</td>
<td>(width of screen)*0.75</td>
</tr>
<tr>
<td><strong>Horizontal position</strong></td>
<td>centered</td>
<td>centered</td>
<td>centered</td>
</tr>
<tr>
<td><strong>Vertical position</strong></td>
<td>centered</td>
<td>(bottom of screen) - 64px</td>
<td>centered</td>
</tr>
<tr>
<td><strong>Depth position</strong></td>
<td>(on top of) none</td>
<td>(on top of) background</td>
<td>(on top of) rotateButton</td>
</tr>
<tr>
<td><strong>Depth attachment</strong></td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td>none</td>
<td>rotateButton.svg</td>
<td>samplePhoto.jpg</td>
</tr>
<tr>
<td><strong>Content layout</strong></td>
<td>none</td>
<td>fit</td>
<td>fit</td>
</tr>
<tr>
<td>Element property</td>
<td>Background</td>
<td>Rotate Button</td>
<td>Photo</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------</td>
<td>---------------</td>
<td>-------</td>
</tr>
<tr>
<td>Direct visual styling</td>
<td>color: #ffffff</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Event trigger</td>
<td>none</td>
<td>manual</td>
<td>co-event: rotateButton</td>
</tr>
<tr>
<td>Trigger delay</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Manual interaction</td>
<td>none</td>
<td>click/tap</td>
<td>none</td>
</tr>
<tr>
<td>State</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Interaction logic</td>
<td>none</td>
<td>button bell-push</td>
<td>none</td>
</tr>
<tr>
<td>Interaction transparency</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Interaction area</td>
<td>none</td>
<td>width * height</td>
<td>none</td>
</tr>
<tr>
<td>Event</td>
<td>none</td>
<td>change of state: Animation</td>
<td>change of state: Animation</td>
</tr>
<tr>
<td>Animation</td>
<td>none</td>
<td>none</td>
<td>rotate: 90 degrees ccw, 300ms</td>
</tr>
<tr>
<td>Animation loop</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Manual moving interaction</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Snap</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Stick scroll</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Scalability logic</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 5.2: Table of three example GUI elements and their properties in terms of the DSL specification
The results of the research into the fundamental building blocks of a functional GUI lead us to conclude that the set of properties we presented in chapter 5 is sufficient to describe a wide range of existing GUIs. Table 5.1 implies that a GUI element can be visually and behaviourally described in 25 main properties. Due to the large sample size of investigated GUIs found in the appendix, it is reasonable to conclude that GUI elements with the stated properties can account for a diverse set of GUIs.

The language design for DSL can be derived from the required visual and behavioural properties stated in section 5.1. Figure 5.4 provides further insight in how these properties are related, easing the understanding of how a converter from DSL to working GUI code should work. Section 5.2 shows that DSL is the only candidate to meet all the requirements for the proposed automated GUI design-to-code workflow and/or be in a current state of maintenance.

An apparent downside to our approach is that maintainability by hand is difficult due to the automation-focused workflow. In addition, it is unclear how much time a software engineer needs to tweak the generated GUI code, compared to implementing the GUI from scratch. The extent to which this proves to be an issue is subject to research as stated below in Future work.

In conclusion, with the requirements for DSL being clarified, the first step of our automated workflow proposal is specified. Thereby, the gap to realise the goal of reducing implementation time and increasing the accuracy of GUI code is narrowing.
CHAPTER 7

Future work

Revisiting the illustration of the proposed workflow, we can assess the future work needed to increase the speed and accuracy of GUI implementation based on a GUI design. Moreover, we state the questions that arise on this workflow and what research is to be conducted.

DSL needs a syntax that meets the requirements and does not deviate unnecessarily from industry standards for best cognition and maintainability. Converters are required to convert design documents into DSL and DSL into GUI code for specific platforms. Last but not least, a certifying application needs to investigate and repair impossibilities and warnings in DSL documents to prevent them from ending up in the final code.

The most important question arising regarding the end result is whether and to what extent implementation time and accuracy will improve. The easiest way to investigate this topic is to implement a single design-to-DSL converter, a single DSL-to-code converter targeting a specific operating system, and a DSL certifier application that sufficiently interprets and resolves errors in DSL documents. Using real-world GUI designs as input to this exercise is of course mandatory to draw representative conclusions.

While the efficiency and the quality of the end result for software engineers is a major motivation for our proposed GUI design-to-code workflow, it is important not to limit the creative possibilities of the GUI designer. If the designer has a 3D interface in mind, an automatic conversion to code through DSL will not work because 3D GUI element properties have not ended up in the list of requirements for DSL. This is an example of why an automated system will
not cover 100% of the possible cases. Therefore, the question arises if this approach reaches a sufficient percentage of unrestricted GUI design-to-code conversions. The sufficiency threshold and whether it is reached or not, completes the set of high-priority future work.


To research the visual and behavioural properties of GUI elements, the following applications have been studied. These applications include desktop, mobile and web applications. Most desktop and mobile applications are native to the macOS and iOS platforms, respectively. Where possible, we used matching counterparts on desktop and mobile to better understand the correlations between the two kinds of platforms. Moreover, we used a wide variety of application categories, such as communication, content creation, content consumption and games.

**Desktop** Calendar, Calculator, Notes, Reminders, App Store, Mail, Messages, Photos, Safari, Pages, Numbers, Keynote, Pixelmator, Clear, Telegram, iA Writer, Skype, Slack, Sketch, Mendeley Desktop, MsPaint, MsWordpad


**Web** theverge.com, ia.net, pixelmator.com, medium.com, atlassian.net, nytimes.com, kickstarter.com