Technical Disclosure Commons

Defensive Publications Series

April 2020

Partial mapping of abstract rich UI model

Nicolas Roard

John Hoford

Partial mapping of abstract rich UI model

ABSTRACT

Visual UI editors generally have certain inflexibilities. For example, if a UI model is used to directly generate compilable code, a developer cannot modify the UI from the generated code. Further, tight coupling that exists between the UI model and the UI elements can make cross-platform development and reactive programming difficult, and can result in limited expressivity. This disclosure describes a rich, abstract, declarative UI model that encompasses layout, navigation, animation, and static visuals. The declarative UI techniques and partial scene mapping described herein can reduce developer workload and enable the building of two-way tooling, cross-platform development, gradual prototype-to-application evolution, separation of static and dynamic content, and dynamic handling of elements mapping.

KEYWORDS

- User interface
- UI editor
- Declarative UI
- Scene mapping
- Rendering engine
- Layout engine
- Motion engine
- UI element
- Draw primitive
- Imperative programming
- Reactive programming

BACKGROUND

User interface builders (also known as visual UI editors) are tools that help developers create their application UI in a visual manner. UI builders typically work on a static model of the UI, enabling developers to have a visual, what-you-see-is-what-you-get (WYSIWYG) representation of the UI during the creation phase, as well as providing drag and drop operations. The UI model can be used by an application generally in two ways: either by generating compilable code from the UI model (approach A, see Fig. 1), or from loading the model at runtime to instantiate the UI elements (approach B, see Fig. 2).

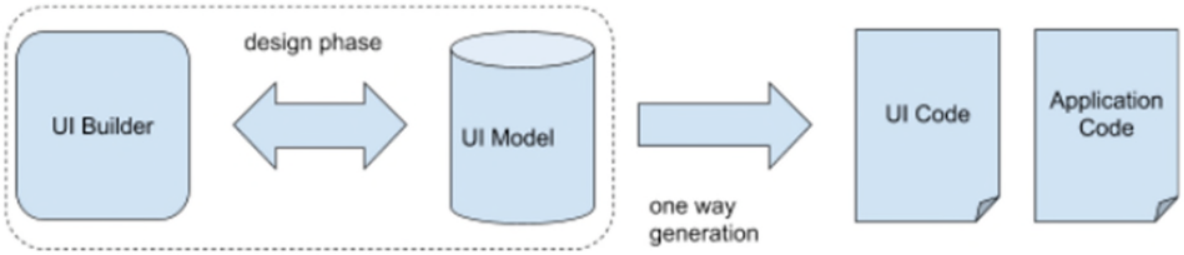

Fig. 1: Approach A, code generation

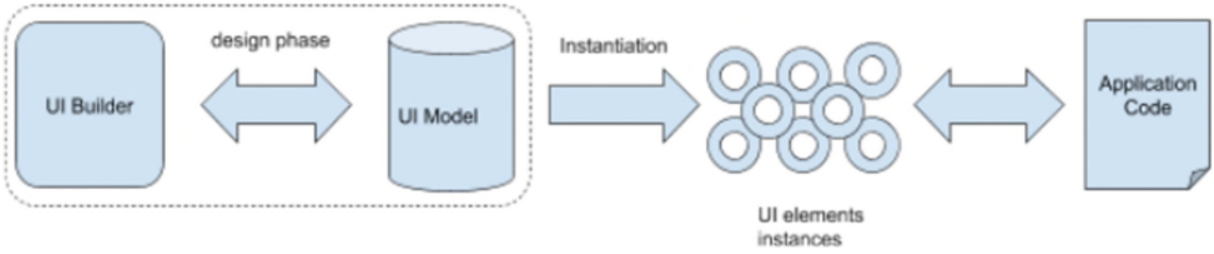

Fig. 2: Approach B, loading model at runtime

UI builders typically operate on a static model describing the UI, as this is easier than operating on the equivalent generated code; the static model has clear limits in what it allows, while the equivalent code representation has larger expressivity.

Both these approaches have issues:

- If the model is used to directly generate compilable code, it is a one-way operation; generally, the developer cannot modify the UI from the generated code using the UI builder tool, thereby limiting the flexibility of the development process.
- In both approaches, there's typically a tight coupling between the model and the final objects representing the UI elements. The coupling is often one-to-one, as shown in Fig. 3.

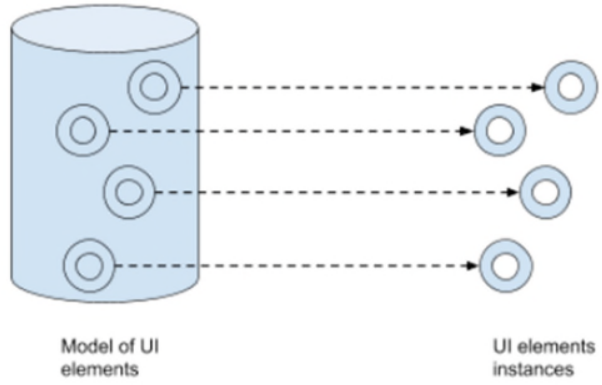

Fig. 3: Tight coupling between model and UI elements

The tight coupling can pose problems for cross-platform applications, where typically separate UIs are created for each platform. It can also show up more subtly by promoting an all-or-nothing model, where defining what should be in the static model versus what should be in code can result in conflicting requirements. Notably, this can make evolving UIs problematic and can render certain patterns like reactive programming difficult to adapt to UI builders.

- A common issue is that the UI models used tend to be relatively simple, typically focusing only on the problem of specifying which UI elements are present on a screen

and how they are laid out (Fig. 4), which covers only a small part of the needs in building modern graphical applications.

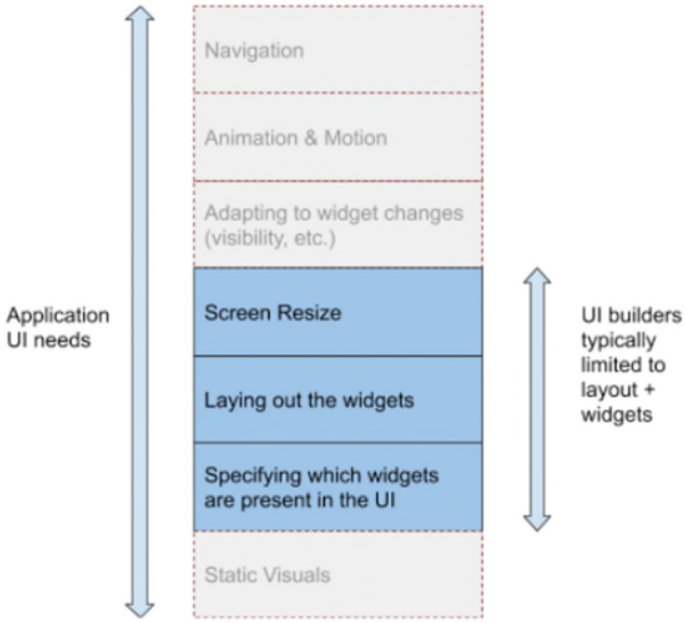

Fig. 4: Limited expressivity of the UI models used by UI builders

UI builders within a reactive programming framework also have problems, as explained below. In an imperative UI programming model, the UI of an application is generally managed using a variation of a three-layers organization (Fig. 5), referred to as model-view-controller (MVC).

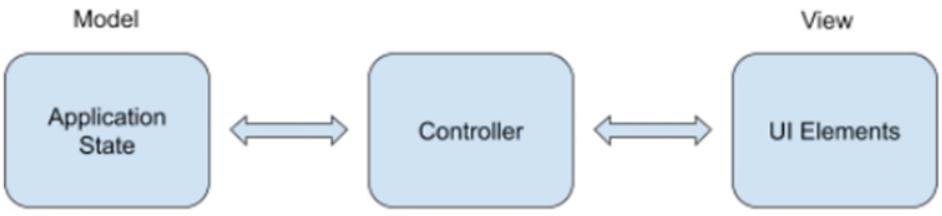

Fig. 5: Typical separation of concerns in traditional UI building

The UI builder in MVC is only concerned with specifying the UI elements, not with how they react to the state; the controller object is the glue, charged with reflecting correctly the application state in the UI and vice-versa. The controller can modify the application state given user inputs coming from the UI elements. This model has several shortcomings, e.g., ensuring consistency between model and view in light of the bi-directionality of data flows.

Fig. 6: Reactive model for managing UI state

In the reactive programming model, managing the state of the UI (reflecting correctly the application state) is instead done automatically (Fig. 6). As such, deciding the UI elements to be displayed (and their state) derives from the application logic. However, this architecture can lead to difficult problems with the integration of visual tooling like UI builders. A common way of mapping UI to state is to specify the UI directly in code.

```
fun MyComponent(name: String, condition: Boolean) {
    if (condition) {
        // create UI A
       Text(text = “Hello $name”)
    } else {
        // create UI B
       Text(text = “Bonjour $name”)
    }
}
```

Fig. 6b: Declarative model for managing UI state in a programming framework

With the UI declared in code (as shown in Fig. 6b), the scenario is similar to approach A (Fig. 1) wherein it can be extremely challenging to build comprehensive UI builders operating on arbitrary code.

DESCRIPTION

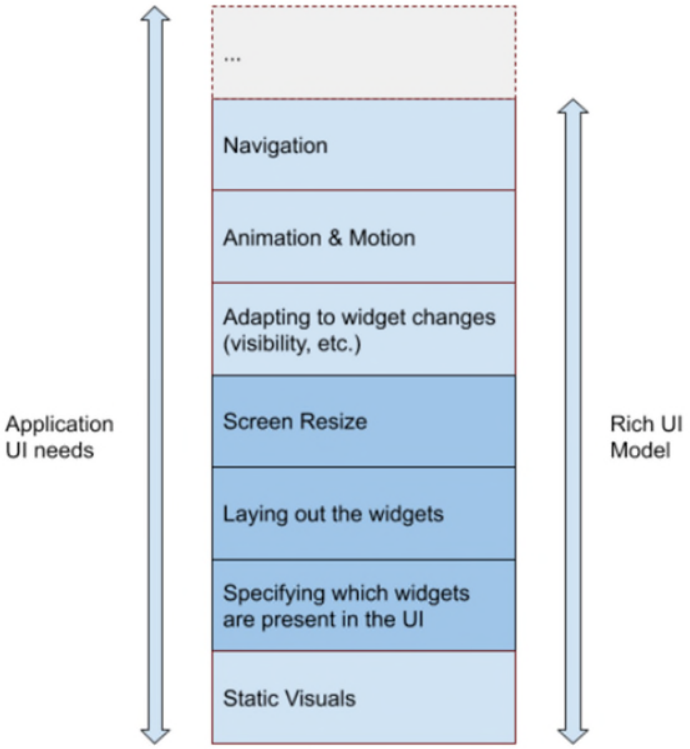

Fig. 7: Rich UI model

This disclosure describes a rich, abstract, declarative UI model, referred to henceforth as "scene," that encompasses layout, navigation, animation, and static visuals (drawings, text, images), as illustrated in Fig. 7. The rich scene model leverages technologies such as declarative UI layout engines and declarative motion and animation engines. Pure static visuals can be handled by the scene engine either by using basic draw primitives of an underlying platform (draw commands, draw text, draw images) or by more powerful rendering engines.

The ability to declaratively specify complex layouts, animation, and motion simplifies the developers' workload in comparison to a fully programmable model. In addition, the declaration of the scene, while still expressive, is done statically, which makes it possible to build fast two-way tooling around it.

Partial scene mapping

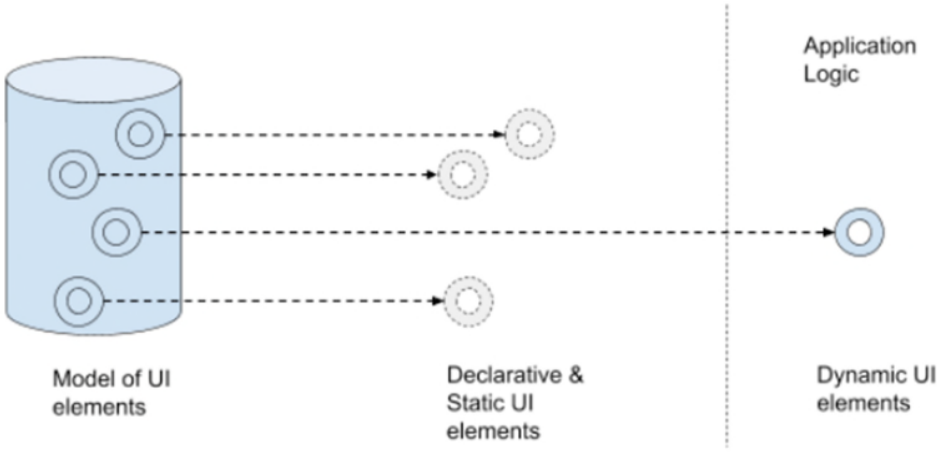

Fig. 8: Partial mapping of UI elements

A scene's elements can be fully or partially mapped at runtime on a given platform. For example, as illustrated in Fig. 8, the only elements that may be mapped to elements in the application logic can be elements dependent on the state of the application. Partial mapping provides several advantages, such as the following.

- *Easier cross-platform support*: Partial mapping can simplify cross-platform UI building by abstracting the layout, navigation, animation, and static visuals instead of specific widgets. This works well in two ways, as follows. The look-and-feel of widgets are typically where platforms differ and existing cross-platform frameworks struggle. In practice, layout, navigation, animation, and visuals comprise a dominant portion of UI and design work.

- *Gradual evolution from prototype to full application*: Allowing partial mapping of the scene encourages gradual construction of the application; it is possible to start with a prototype of an app and gradually transition it to a fully working app. As an example, a custom component can first exist simply as a static visual representation until the programmer maps it to a real implementation. As a rich model, the scene can also provide example or sample data to be used until the developer provides real data.
- *Better separation of static and declarative content*: Partial mapping of the scene enables the separation of static and declarative content in the scene outside of the application logic. In this manner, the application side can be simplified and can focus on logic and state management. This maps well with the reactive programming approach, although it is also applicable to the imperative UI programming approach. In the reactive model, this means that instead of having to declare everything (static elements and state-dependent elements), a developer can concentrate on specifying only the state-dependent elements.
- *Better dynamic handling of elements mapping*: As elements of the scene may or may not be mapped to concrete UI elements, greater flexibility can be expressed directly in the scene. For example, the behavior on mapping or non-mapping can be specified, e.g., if not mapped, don’t appear; or conversely, do appear. This implies, for example, that small changes in the UI (such as a section of the screen appearing/disappearing depending on the state) can be easily handled without having to rebuild a completely separate scene.

Example: Application to a UI programming framework

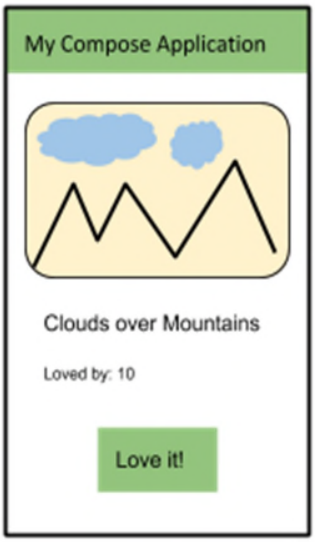

Fig. 9: A programming framework with code declaration of UI

Fig. 9 illustrates an example of a screen created in a reactive programming framework, with the corresponding implementation with the UI defined in the pseudocode illustrated in Fig. 11.

```
function NewsStory() {
    val image = +imageResource(R.drawable.header)
    var loves = +state { 0 }
    Theme {
            DrawImage(image)

            HeightSpacer(16.dp)

            Text("Clouds over Mountains",
                    style = (+themeTextStyle { h6 }).withOpacity(0.87f))
            Text("XYZ, California, Dec 2019",
                    style = (+themeTextStyle { body2 }).withOpacity(0.87f))
            Text("Loved by: ${loves.value}")
            Button("Love it!", onClick = { loves.value++ })
        }
    }
}
```

Fig. 10: Code declaration of UI within a programming framework

An example of how a partial mapping of a scene can be done is illustrated in the pseudocode of Fig. 11, which shows that the scene is defined in a separate file (`R.scene.news_story`).

```
function NewsStory() {
    var loves = +state { 0 }
    Scene(R.scene.news_story) {
        Text(modifier = Tag(“label”),
             "Loved by: ${loves.value}")
        Button(modifier = Tag(“loves_button”),
             "Love it!", onClick = { loves.value++ })
    }
}
```

Fig. 11: Partial mapping of the scene within a programming framework

Fig. 11 illustrates how the techniques of this disclosure can reduce the number of lines of code necessary for this screen without losing any of the application-state management advantages of the programming framework.

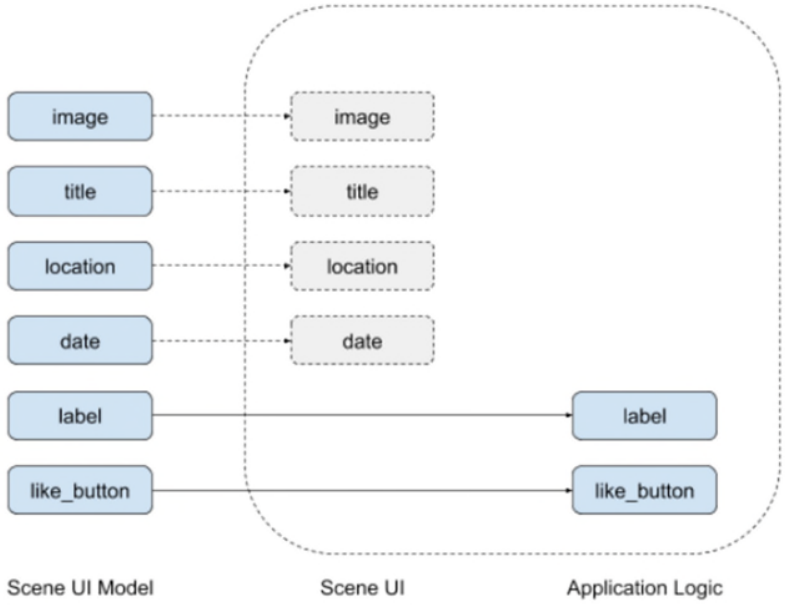

Fig. 12: Partial mapping of the scene within a programming framework

As illustrated in Fig. 12, the non-dynamic elements (images, static text…) have been left in the scene, and the code only needs to provide elements that depend on the current state of the application and map them to their placeholder in the scene description. If at some point, some of the elements need to be set or defined dynamically, the evolution path is seamless and gradual, without having to modify the scene.

CONCLUSION

This disclosure describes a rich, abstract, declarative UI model that encompasses layout, navigation, animation, and static visuals. The declarative UI techniques and partial scene mapping described herein can reduce developer workload and enable the building of two-way tooling, cross-platform development, gradual prototype-to-application evolution, separation of static and dynamic content, and dynamic handling of elements mapping.

REFERENCES

1. Hayton, Richard, and Dave Otway. "Methods and apparatus for communicating changes between a user-interface and an executing application, using property paths." U.S. Patent Application 11/565,923, filed December 01, 2006.
2. Beda, Joseph, Kevin Gallo, Adam Smith, Gilman Wong, and Sriram Subramanian. "Markup language and object model for vector graphics." U.S. Patent Application 10/693,633, filed October 23, 2003.