Evolution of Functional UI Paradigms

Michael Sperber
michael.sperber@active-group.de
Active Group GmbH
Tübingen, Germany

Markus Schlegel
markus.schlegel@active-group.de
Active Group GmbH
Tübingen, Germany

Abstract

Functional paradigms for user-interface (UI) programming have undergone significant evolution, from early stream-based approaches, monad-based toolkits mimicking OO practice to modern model-view-update frameworks. Changing from the classic Model-View-Controller pattern to functional approaches drastically reduces coupling and improves maintainability and testability. On the other hand, achieving good modularity with functional approaches is an ongoing challenge. This paper traces the evolution of functional UI toolkits along with the architectural implications of their designs—including two of our own—and summarizes the current state of the art and discusses remaining issues.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Software architectures**; **Functional languages**; **Development frameworks and environments**.

Keywords: functional programming, software architecture, patterns, graphical user interfaces

ACM Reference Format:
Michael Sperber and Markus Schlegel. 2025. Evolution of Functional UI Paradigms. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3759163.3760429

1 Introduction

The architecture of applications with graphical user interfaces (GUI), has been based on the Model/View separation for for almost 50 years. Yet, the architecture of these applications and their toolkits continues to evolve. This paper examines the evolution of UI toolkits and application architecture since their inception from a functional perspective.

Overview. Section 2 summarizes the software architecture concepts relevant for the discussion in this paper. Section 3 reviews the Model-View-Controller pattern, the starting point for all modern UI toolkits. Section 4 discusses the event loop, an implementation aspect of many UI toolkits. Section 5 reviews early functional UI toolkits. Section 6 describes a transition point between these early toolkits and the toolkits of the modern era—Racket's Universe teachpack. Section 7 discusses Elm and React, two implementations of the functional Model-View-Update pattern. Section 8 briefly reviews modern object-oriented toolkits for comparison. The Model-View-Update pattern still struggles with modularity issues described in Section 9. Section 10 describes the Reacl toolkit, designed to tackle these issues at a technical level. Section 11 explains why UI programming remains a struggle despite all the technical progress, and Section 12 describes the architectural approach that we have developed to deal with it. Section 14 concludes.

2 UI and Software Architecture

Starting with the *Model-View-Controller* pattern (MVC) [21] in the 1970s, the software industry and research community have produced a plethora of *UI toolkits*, libraries that provide the conceptual elements of a UI—text, input fields, buttons, lists, menus, grids and more—as objects to be created and manipulated by the program, henceforth called *widgets*. Each toolkit dictates or at least constrains the organization of the applications that use them.

The job of a UI toolkit is to enable developers to create and maintain UI applications, but these dual requirements—fast creation and effective long-term maintenance—tend to be in conflict. This is especially pernicious in practice, as most of the cost produced by a software system is in maintenance, not initial creation [24].

This phenomenon seems to be especially pronounced in GUI applications: The Visual Basic 6 (VB6) environment for building GUI applications [4] was optimized for fast initial creation, and as such, it was one of the most successful such environments of all time. Microsoft relegated VB6 to "legacy" status in 2008. Despite this, many VB6 applications are still around at the time of writing, even though their further development is painful and costly, and their maintainers understand the importance of moving away from an unsupported platform.

Moving away from VB6 is so difficult because a VB6 application typically combines UI-related code, domain logic, and

FUNARCH '25, Singapore, Singapore

ACM ISBN 979-8-4007-2146-5/25/10
https://doi.org/10.1145/3759163.3760429

possible interactions with a database in the same function, making it all but impossible to replace any of these parts of the system independently. This effectively makes a "big-bang migration" the only path away from VB6, but this is often economically infeasible.

The problem of VB6 projects and many other projects with high maintenance costs is *coupling* [24], dependencies between different conceptual building blocks of the software. Coupling causes changes in one place to require changes elsewhere. Consequently, the central concern of software architecture is to enable effective long-term maintenance by minimizing coupling. A software project can control coupling through modularization [19]—splitting the project into building blocks and maintaining strong boundaries—"modularity"—between them.

The shared concern of UI applications is to establish modularity between the UI code and the domain logic. For the rest of this paper, we will examine UI toolkits and paradigms with respect to these tenets of software architecture: How do they impact coupling, modularity, and thus maintainability?

3 Model-View-Controller

MVC is the ancestor of most contemporary UI paradigms and frameworks. The original goal of this pattern was to enable a particular kind of change—changing or replacing the UI without changing the domain logic. To that end, the central innovation of MVC was the decoupling of UI code ("view") from the domain logic ("model").

The decoupling between view and model also has other practical implications, such as for automatic tests. Environments that strongly couple view and model, such as VB6, make applications difficult to test, as the only way of invoking domain functionality is via the UI. Often, the only resort in these cases is to simulate a user programmatically, which is expensive and brittle.

MVC programs need to keep the view current when the state of the model changes. To that end, models typically implement a variation of the *observer pattern* [15]: The model maintains a list of "dependents", objects that it notifies on changes by calling an `update` method on them. (The controller is tightly coupled to the view, and its role is unimportant for this discussion.) As the `update` method informs the view of state changes, this approach is inherently imperative. MVC has nice modularity properties, as each part of the model only needs to be coupled to "its" part of the view, and composition is easy. It also creates challenges to software architects:

Update Keeping the view current with respect to the model involves two tasks: initially *constructing* the view by creating the various UI widgets, and later *mutating* the view upon calls to `update`. Conceptually, the result of updating the view should be the same as re-constructing the view, but the concrete code for both is quite different, implementing the same logic twice.

Modularity To achieve modularity, architects should be able to split a large, complex view into loosely coupled subviews. This raises the question of how big those subviews should be. Making them large and correspondingly represent a larger chunk of the model simplifies the change logic, as it requires fewer implementations of `update`. However it also increases coupling between model and view. It also causes potential performance issues when a change is only to a small part of the model and only needs a small part of the view to change: In that case, `update` either spends time drilling down to this small part of the view or changing larger parts of the view than necessary.

Circularity The view typically includes interactive elements that cause changes in the model. Naively implemented, these changes indirectly trigger calls to `update` in the view, which again might spill over into changes to the model, causing a cyclic call chain.

Consider the following practical example: Figure 1 shows object-oriented pseudocode for a weather model that contains air pressure and temperature, each with its own encapsulated state. Figure 2 shows skeleton code for a view that displays both in text fields. The code demonstrates the modularity of the approach, as `PressureView` and `TemperatureView` each subscribe to their respective models, with no coordination required from `WeatherView`.

The example also illustrates the first two challenges: The code for constructing the string representing the pressure and temperature is duplicated between the code that constructs the text field and the code that updates it. Some code can be abstracted, but the fundamental *Update* challenge remains. The code contains two sub-views, updated separately, addressing the *Modularity* challenge. Alternatively, the subscription and update could happen between `WeatherView` and `Weather`, reducing modularity somewhat but also reducing the amount of code required.

The views are missing code to unsubscribe them from their respective models—this would further complicate the interaction between model and view.

Figure 3 shows a different view, just on the temperature, but with two sub-views showing the temperature with different units. The code also adds the element of interactivity: When the user interacts with the view, the two observers attached to the views update the model. This demonstrates the *Circularity* challenge described above: There is now an update loop between the model and the two views: Each update to the model updates the view, which updates the model, and so on. The code could break the loop by checking whether the new value is different from the old, but this is brittle—especially as floating-point rounding is involved in the conversion between the three units.

```
class Weather(field pressure: Pressure,
              field temperature: Temperature)
{ ... }
interface Observer {
  method update(info)
}
class Model {
  var observers: List Observer

  method changed(info) {
    for each observer in self.observers
      observer.update(info)

  method subscribe(observer) {
    observers += observer
  }
}
class Pressure extends Model {
  var hPa: double
  method setHPa(newHPa: double) {
    self.hPa = newHPa
    self.changed(newHPa)
  }
}
class Temperature extends Model {
  var kelvin: double
  method setKelvin(newKelvin: double) {
    self.kelvin = newKelvin
    self.changed(newKelvin)
  }
}
```

Figure 1. Model for weather data

```
class TemperatureView
  (field temperatureModel: Temperature) {
  var textField =
   new TextField(toText(temperatureModel.kelvin)
        + "K")

  temperatureModel.subscribe(self)

  method update(newKelvin) {
    textField.setText(toText(newKelvin) + "K")
  }
}
class PressureView
  (field pressureModel: Pressure) {
  var textField =
    new TextField(toText(pressureModel.hPa)
        + "hPa")

  pressureModel.subscribe(self)

  method update(newHPa) {
    textField.setText(toText(newHPa) + "hPa")
  }
}
class WeatherView(field weatherModel: Weather) {
  var temperatureView =
   new TemperatureView(weatherModel.temperature)
  var pressureView =
    new PressureView(weatherModel.pressure)
  ...
}
```

Figure 2. View for weather data

4 The Curse of the Event Loop

MVC UI toolkit also face a recurring implementation challenge: While a MVC program constructs the UI in terms of hierarchically organized views, it must display the UI as a flat panel of pixels. This means that technically users interact with the UI panel as a whole, and the UI toolkit must infer the specific sub-view that is the target of the interactions. Consider for example a button, which the user presses by clicking with a mouse: The UI toolkit must infer from the position of the mouse cursor the particular button view at those coordinates, and cause its subscribers to be notified.

UI toolkits have traditionally chosen to implement this process using an *event loop*, a piece of code that receives hardware input events and calls subscriber callback of the corresponding views. This happens repeatedly *ad infinitum*, hence the "loop". Here is pseudocode for a typical event loop:

```
while (true) {
  var event = get_input_event()
  dispatch_event(event)
}
```

For this to work, the `update` methods of the view subscribers must not block, as this would prevent further user interaction. If an `update` method needs to perform I/O or other blocking operations, this restriction effectively inverts the flow of control: The method must start the operation asynchronously and register it with the event loop, to be notified when the operation has completed, which is often awkward.

In principle, UI programs could also just start threads that perform blocking operations synchronously, but interact with the UI asynchronously. However, many UI toolkits insist on their functions being called from the thread the UI toolkit runs in, further complicating matters.

Thus, typical MVC UI toolkits couple the control flow of the central event loop to the control flow of the subscriber callbacks, which includes the code that manipulates the model. The restrictions resulting from this coupling often force developers to write these callbacks and their dependencies in such a way that their arrangement in the code does not follow the actual sequence of events at run time, creating a style of programming colloquially known as *callback hell*.

```
class TemperatureView2
  (field temperatureModel: Temperature) {
  var celsiusView =
    new TextField
      (toText(kToC(temperatureModel.kelvin)))
  var fahrenheitView =
    new TextField
      (toText(kToF(temperatureModel.kelvin)))

  temperatureModel.subscribe(new Observer {
      method update(newKelvin) {
     celsiusView.setText(toText(kToC(newKelvin)))
    }
  })
  temperatureModel.subscribe(new Observer {
      method update(newKelvin) {
     fahrenheitView.setText(toText(kToC(newKelvin)))
    }
  })
  celsiusView.subscribe(new Observer {
     method update(...) {
      temperatureModel.setKelvin(cToK(...))
     }
  })
  fahrenheitView.subscribe(new Observer {
     method update(...) {
      temperatureModel.setKelvin(fToK(...))
     }
  })
}
```

Figure 3. Two different sub-views on the same model

5 Functional UI Toolkits: The Early Days

This section briefly describes some early UI toolkits for functional languages. We focus on toolkits that rely on pure functions to implement the UI, and omit those that basically provide bindings for underlying C/C++ libraries and thus inherit its imperative MVC pattern. We also omit UI toolkits that only cover certain kinds of applications, like Eros [11] which is for UIs that construct values.

5.1 eXene

The eXene [16] toolkit was developed as part of the Concurrent ML project [22] (CML). eXene uses Concurrent ML to solve the event-loop problem described in Section 4. In eXene, the global event loop is relegated to the background in favor of individual event loops running in threads, each associated with a particular UI widget. Rather than calling subscriber methods directly, the global event loop sends asynchronous messages to the widget event loops, decoupling them. Developers are afforded great flexibility in orchestrating their actions through CML's event combinators.

While CML and eXene originated in the context of functional programming, the overall approach of UI manipulation in eXene remains firmly imperative. Thus, eXene inherits the *Update* challenge from Section 3.

5.2 Fudgets

While eXene's Standard ML is (also) an imperative language, Haskell's lazy evaluation makes just adding functions that "do stuff" impractical. Consequently, interacting with the outside world was a challenge for the designers of Haskell during its nascency in the 1990s. This also affects the designers of UI libraries. A notable early attempt to do this was the *Fudgets* toolkit [5]. (Fudgets is still being maintained.)

Fudgets represents a UI component (a *fudget*) as a *stream processor* of type `F a b` that transforms a stream of input values of type `a` into output values of type `b`. Fudgets can be combined into sums and products, building UI structure and reactivity in a hierarchical fashion. Fudgets completely abstracts the event-loop paradigm away from the user program, which consists exclusively of pure functions.

Most UIs need to have dynamic structure—for example, when displaying a list of UI items that supports adding and removing elements. Fudgets addresses this by combinators that accept special command values describing mutations of the UI structure, effectively sneaking in imperative elements in its otherwise pure model. Thus, Fudgets also partially inherits the *Update* challenge from Section 3.

5.3 Fruit

Fruit [7] re-imagines the basic idea of Fudgets on top of the idea of Functional-Reactive Programming [10], which had been developed in the meantime. Fruit retains the purity and compositionality of Fudgets, replacing the stream processors of Fudgets with FRP signal transformers. There are two notable differences between Fudget and Fruit, however:

- Fudgets widgets have implicit access to interactive events, while Fruit requires them to be threaded explicitly through the program. This means that a Fruit program is coupled to all aspects of the "outside world" it interacts with, making it difficult to add interactions (such as file or network I/O) after the fact.
- Another difference concerns dynamic UI components: Whereas Fudgets views display the values coming from a stream (and are thus "outside" those streams), Fruit has the views themselves be the values of the FRP signals, and dynamic structure can be expressed by just having new values return a different view.

Thus, Fruit avoids the *Update* challenge from Section 3, but comes with its own modularity problem. The Yampa framework [20] builds on similar ideas, but represents reactivity on *monadic stream functions* that are parameterized over a monad, effectively re-introducing imperative programming to the paradigm.

5.4 Haggis

Haggis was a short-lived UI toolkit for GHC [13], which took a middle path between eXene and Fudgets: While Fudgets provides a purely functional UI for constructing widgets via explicit combinators, Haggis provides an imperative UI running in the `IO` monad for this construction. Haggis deals with the event loop similarly to eXene, replacing Concurrent ML with Concurrent Haskell. For the reactive part of a UI widget, Haggis provides a handle separate from the view part. The program can wait for interaction on these handles also in the `IO` monad (in separate threads if that is convenient), and allows combining them in a way similar to CML events. Thus, Haggis also inherits the *Update* challenge from Section 3.

6 The Racket Universe Teachpack

Teachers of introductory programming courses would often like to enable their students to write interactive programs. The complexity of UI toolkits have traditionally made this difficult, at least in the era of toolkits described in the previous section. Since 2003, the Racket system has come with a library called the *Universe teachpack* for an introductory course [12] that allows writing distributed, interactive video games in a purely functional manner. We focus here on the "world" part of the Universe teachpack, i.e. video games without distribution.

The Universe teachpack is basically "half" a UI toolkit. It is an interesting subject for study, as it does address the pernicious *Update* challenge of the UI toolkits described above. To implement such a video game, the developer only has to implement a small number of pure functions, all of which operate on a model represented by a single value of type `WorldSt`, which is entirely under the control of the program:

```
on-draw : WorldSt ⟶ Image
on-tick :  WorldSt ⟶ WorldSt
on-key :  WorldSt KeyEvt ⟶ WorldSt
on-mouse : WorldSt Nat Nat MouseEvt ⟶ WorldSt
```

These functions are all callbacks for the Universe teachpack, programs contains no explicit calls to them. The `on-draw` function handles view construction and uses Racket's `image` teachpack to build a single frame of the video game. (The `image` teachpack is a combinator library in the style of Henderson's functional geometry [18].) There is no separate function for view *update*: Whenever the model transitions to a new state, the Universe teachpack simply calls `on-draw` again and redraws the *entire* frame. This is fast enough in practice to achieve reasonably fluid animation.

The other functions provide reactivity to different kinds of events: All of them take a `WorldSt` state as input and output a new state, which the teachpack feeds into `on-draw`. `On-tick` is called on every clock tick to provide movement. `On-key` is called on key presses, and `on-mouse` on mouse events. Writing these functions is well within the purview of introductory-course students.

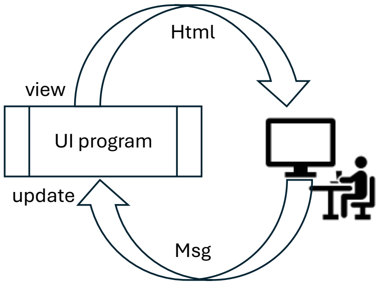


Figure 4. The Model-View-Update pattern

Consequently, the Universe teachpack uses a "brute-force" approach to the view-update problem: Use a single monolithic value for the model state, and redraw everything every time the model moves to a new state. This means no separate view-update code—*Update* challenge solved.

Unfortunately, this approach also abandons the good modularity properties of object-oriented MVC toolkits: From the point of view of the UI toolkit, the model state is a single value, and there is no simple way to split it—and to split the corresponding view elements into sub-views. Thus, the Universe teachpack does not address the *Modularity* challenge from Section 3. This is not a problem in practice, as the video games implemented in a class are simple enough that unified state is appropriate. However, it does prevent easily scaling the approach to realistic UIs in a modular fashion.

7 The Evolution of Elm and React

Elm [9] is a UI toolkit and companion Haskell-like language that translates to JavaScript and thus allows implementing web applications. Elm is in production use in open-source and commercial projects. Elm's initial release in 2012 was based on FRP Signals, and was similar to Fruit. Elm eventually evolved to a different approach, leaving its FRP legacy behind in 2016 [8]. This approach gives rise to an alternative UI pattern called *Model-View-Update*. An Elm program mostly consists of only two pure functions operating on values of a type `Model`, defined by the UI program:

```
view : Model -> Html Msg
update : Msg -> Model -> (Model, Cmd Msg)
```

Figure 4 illustrates the pattern. The `view` function constructs the UI, represented as a HTML tree. Each interactive UI widget carries a message of the `Msg` type (also defined by the UI program) that is generated when the user interacts with the widget. The Elm toolkit then passes this message to the `update` function that generates a new model, which again gets fed into `view` and so on.

The `Cmd` type allows triggering impure actions outside of user interactions. In particular, a program performs network requests by issuing commands, which get executed asynchronously and result in incoming messages, thus ameliorating issues with JavaScript's inherent event loop.

View-Model-Update is simple and easy to learn, and does not require the developer to know higher-level abstractions like signals, stream transformers or monads. It also avoids the circular dependency between view and model.

Model-View-Update is quite similar to Racket's Universe teachpack: The view is re-constructed after every user interaction, and it uses a single, monolithic value for the model state. The possible messages belong to a single central type. Consequently, Elm also inherits the Universe teachpack's architectural trade-offs: It solves the *Update* challenge from Section 3, but has no notion of an independent, composable "UI component" and thus faces the *Modularity* challenge.

Shortly after Elm's initial release, Facebook released their web UI toolkit *React*, which was also based on a variation of the Model-View-Update paradigm. React's `render` function corresponds to Elm's `view`: It transforms the model state into a HTML tree. Unlike Elm, React has a notion of a UI component—a sub-view with its own associated state.

Where Elm gives the UI program complete freedom in the choice of `Model` type, React differentiates two kinds of state: Immutable *properties* flow downward in the sub-view tree—the property of a sub-view needs to be extractable purely from the property of its parent. Properties face the *Modularity* challenge. Each component also has its own associated mutable *state*, which needs to be a JavaScript hash map.

Thus, React's model of reactivity is different from Elm's: The UI program attaches imperative callbacks to UI components, and these callbacks can either mutate the component state or cause the properties to be replaced. Conceptually, React ignores the difference between these two methods and re-renders the entire UI on each interaction, just like Elm.

Both React and Elm leave solving the *Modularity* challenge to the developers, and the communities of both have produced patterns to alleviate the problem.

Elm and React and the Model-View-Update pattern represented an erstwhile peak in the evolution of functional APIs, intellectually hailing from Fudgets and Fruit. Consequently, we take Model-View-Update as the starting point for the further discussion.

8 UI Toolkits Elsewhere

Evolution did not just happen in functional approaches to UI programming. Object-oriented UI toolkits have also evolved, specifically those for web applications. Modern OO toolkits like Angular, Svelte, and Vue.js have also attempted to solve the *Update* challenge from Section 3 using a common approach (also available with React).

These toolkits rely on the same underlying architecture as the original MVC pattern and require the program to construct the UI initially and update specific parts of the UI corresponding to specific changes in the model. This way, they avoid the overhead of Model-View-Update from (conceptually) re-constructing the UI on every change.

However, instead of requiring the programmer to implement the update logic manually, these toolkits feature a pre-processor that generates the update code automatically from the construction code. This convenience comes at a price, as these toolkits require the code changing the state to be exposed to the UI toolkit, and for that code to use conventions recognizable to the pre-processor. This means that the model cannot be implemented independently of the UI toolkit, and is thus coupled to the view—effectively giving up the Model/View separation, the central tenet of MVC.

9 Functional Modularity Challenges

Taking Elm's and React's Model-View-Update pattern as a starting point, we summarize the architectural tradeoffs listed in Section 3:

- Model-View-Update solves the *Update* challenge: A single function describes view construction, no separate logic for view update is needed.
- The *Modularity* challenge—implementing modular sub-views—remains: A functional Elm/React application manages its model state as a single mutable reference at the top level, with everything below managed functionally.
- Model-View-Update also solves the *Circularity* challenge, as the cyclic call chain is broken by the succession of models and updates presented to the UI toolkit.

Solving the *Modularity* challenge in the context of functional programming means establishing a notion of "UI component" without resorting to imperative state updates. A solution approach needs to address the facets described in this section: Updating the model state in a modular fashion, dealing with UI-local state, and breaking up global dispatch.

We illustrate these facets using a simple example: an application that manages phone-book entries with name and phone number. The application manages the phonebook in a functional manner, as a functional list of immutable entry records. Its UI contains a sub-view component that allows viewing and editing a single name within a text field, along with a "Submit" button for actually changing the phonebook.

9.1 Global Model State Update

When the user changes the name, this is a local change from the point of view of the UI component, but the application must update its global phonebook reference. Thus, updating the state is not the job of the name UI component but of all the components above it in the hierarchy that need to re-construct the state with the one changed name, and finally change the global reference at the top.

9.2 UI-Local State

The model-state update happens when the user clicks the "Submit" button. However, the UI has to manage the contents of the text field as the user types in the new name one letter at a time. This intermediate state is of no interest to the model, hence it is "UI-local" and not an aspect of the model. Thus, the UI toolkit should enable separating the model state from the UI-local state.

Note that the distinction between model state and UI-local state depends on context and judgement: The name UI component itself has two sub-components, a text field and the submit button. The text field's model will usually be the visible text. However, embedded within the name component, the text field's model state becomes the name component's UI-local state.

9.3 Global Dispatch

In Elm, the UI program communicates requests to change the model state (as well as other side effects) via messages of type `Msg`, which the toolkit subsequently passes to the `update` function. This `update` function is global, and thus unmodular. Making the global dispatch modular requires associating it with an individual UI component rather than the program as a whole.

However, a UI component is often not able to handle the messages generated by user interaction with its view. Consider again the name UI component: The name should probably not be empty, and thus the "Submit" should be deactivated as long as the text field is empty. Thus, the text-field component must pass a message upwards to its enclosing name component, informing it of the state change. Correspondingly, the enclosing component must be able to handle the message coming up from its sub-component.

Furthermore, the type of the message produced by the sub-component must match the type expected by the enclosing component. This is not necessarily the case when the two are developed separately: For this case, the UI toolkit must offer a mechanism to transform the message along the way from its producer to its handler.

10 From Model-View-Update to Reacl

As a response to challenges described in the previous section, we started in 2014 to work on our own UI toolkit for ClojureScript, *Reacl* [1]. Reacl is similar to Halogen, a UI toolkit for PureScript [17], even though both were developed independently. This section describes how Reacl tackles the modularity challenges, using the phonebook example. Reacl evolved significantly over its lifetime, eventually leading to its successor reacl-c described in Section 12. We describe Reacl version 2 from about 2019.

The phonebook example starts with a namespace declaration that imports both Reacl and the record library from *Active Data* [3], a library for making data modeling explicit:

```
(ns phonebook
  (:require
    [reacl2.core :as reacl :include-macros true]
    [reacl2.dom :as dom :include-macros true]
    [active.data.record :as record
      :include-macros true
      :refer-macros (def-record)]
    [active.data.realm :as realm]))
```

We use Active Data to define a record type for an entry in the phonebook, the central aspect of the model:

```
(def-record entry
  [entry-name :- realm/string
   entry-phone-number :- realm/string])
```

In this declaration, `entry` is both the record type and constructor, `entry-name`, `entry-phone-number` each double as field name and getter/selector. The `realm/` annotations declare the run-time types of the fields of the record.

We now construct a UI for editing the name of an entry. To that end, we start with a reusable component for an editable text field. This is implemented via a *class* in Reacl that defines how the text-field components are rendered and behave:

```
(def-record change-text
  [change-text-text :- realm/string])

(reacl/defclass text-field this text []
  render
  (dom/input {:onchange
              (fn [e]
                (reacl/send-message!
                 this
                 (change-text
                   change-text-text
                   (.. e -target -value))))
              :value text})
  handle-message
  (fn [msg]
    (cond
      (record/is-a? change-text msg)
      (reacl/return
        :app-state (change-text-text msg)))))
```

In this declaration, `this` is the name that the code can use to refer to the current component (similar to `this` in Java), and `text` is the *app state* of the component—its model.

Two clauses follow, the first of which is `render`, which generates the view from the app state `text`. This is straightforward HTML with a callback attached that fires whenever the user changes the text. The callback sends a message to the component that describes the user's intention, in this case to change the text in the form of a `change-text` record.

Components can send messages to arbitrary other components, but it is typical to send them "to themselves," as is shown here. The `handle-message` clause defines how a component reacts to a message it receives. In this case, the only possible message is a `change-text` record, but often components allow for more than one kind of interaction, which is why there is a cond here that identifies the `change-text` record. The function then calls `reacl/return`, which tells

Reacl what to do—in this case replace the app state of the component with the text from the message, performing an update as in the Model-View-Update pattern.[1]

In principle, the callback attached to the text field could itself perform the update—first creating and then dispatching on a message is more work. However, this separation of concerns makes the handle-message function testable, even outside of the running UI application. (The render function is also testable by inspecting the returned HTML, but this comes with caveats described in Section 11.)

Next, we create a UI component for actually editing a name from a phonebook. The idea is that the phonebook will only be changed when the user clicks on a Submit button. Consequently, the app state of the name component will only change when Submit is pressed. However, the component still needs to keep track of current content of the text field. To that end, it keeps a *local state*, which is not reflected in the model state. The class starts like this:

```
(reacl/defclass text-editor
  this text []
  local-state [current-text text]
```

Here, text is the app state of the component. The local state is called current-name and its initial value upon creation of the component is text. Here is the render clause, which creates an HTML form:

```
render
(dom/form
 {:onsubmit
   (fn [e] (.preventDefault e)
           (reacl/send-message! this (submit)))}
 (dom/div (text-field
           (reacl/opt
             :reaction
             (reacl/reaction
               this
               (fn [text]
                 (change-text
                   change-text-text text))))
           current-text)
          (dom/button "Submit")))
```

This calls text-field to create a text-field component, passing it a *reaction* and current-text as its app state. (This shows that one component's app state can be another component's local state.) The reaction defines how the text-editor component reacts to changes in the text-field component's app state (passed in as text to the function), typically by sending a message. Here, the reaction sends a change-text message (re-used from text-field) to this.

The form callback, which gets called when the user clicks Submit, sends a singleton submit message as per this record declaration:

```
(def-record submit [])
```

(The .preventDefault DOM call is just a technical necessity.)

Finally, the handle-message function dispatches on the two messages, updating the local state for change-name and the app state for submit:

```
handle-message
(fn [msg]
  (cond
    (record/is-a? change-name msg)
    (reacl/return
      :local-state (change-name-text msg))
    (record/is-a? submit msg)
    (reacl/return :app-state current-name))))
```

This example shows how Reacl deals with the three challenges described in Section 9:

Global Model State Update A handle-message function only describes how to update its own model/app state, with no knowledge of and therefore no coupling to the global state of the application.

UI-Local State The local-state mechanism keeps local state out of the model.

Global Dispatch The handle-message function is per component class, no global dispatch is necessary.

It should be noted that Reacl also has a mechanism not described here for side effects external to the UI component such as network communication. In order to make the core of the UI application pure, it creates an *action*, which is similar to a message with the difference that it does not specify a receiver but gets propagated upwards in the tree formed by the components until a component either handles it or transforms it into an action understood further up. This achieves modularity for actions in a way similar to reactions.

11 Inherently Vague Semantics

User interface construction remains a struggle. In addition to the challenges described above, UI programming—and testing—is difficult for reasons beyond the grasp of UI tools.

Model-View-Update should make UI tests easy: Run the view function for a sample state of your domain logic and compare the resulting graphical display (view state) to your desired version. Even forgetting about the dynamic nature of UIs for a moment, this approach rarely results in good tests. Take this temperature display in pseudocode:

```
(defn view [temp]
  (dom/div
    {:style (if (too-hot? temp)
              "background: red;" "")}
    (temperature->string temp))
(assert-equal (view 22) (dom/div "22")
(assert-equal (view 183)
  (dom/div {:style "background: red;"} "183"))
```

[1] Since the development of Reacl, React has adopted a similar mechanism to handle-message called *reducers*.

This trivial test is problematic if we want to emphasize not with a red background but a border. Our test demands that the background be red. To make the change, we have to adapt both our implementation *and* our test.

A good test checks whether an implementation satisfies a given specification. Howeber, the specification of a UI components is hard to separate from its implementation. In this case, we want to ensure "emphasis", which is hard to formalize. Moreover, users' habits and expectations change: An emphasis today might have been overlooked ten years ago. Additionally, context is important. A graphical representation that is emphatic when surrounded by white space may appears too timid surrounded by loud advertisements.

Lehman's SPE classification [6] distinguishes programs depending on how well specified and specifiable they are. *S* is for "programs whose function is formally defined by and derivable from a specification", *P* is for programs that have a formal specification, but "the acceptability of a solution is determined by the environment in which it is embedded", and *E* is for "programs that mechanize a human or societal activity". Classes *P* and *E* are further aggregated to form class *A*—programs "that represent a computer application in the real world". The distinction between *S* and *A* programs aligns with our distinction between software aspects that have a precise formal semantics and vague aspects like emphasis. *S* programs are amenable to automated tests or even formal verification. *A* programs on the other hand require manual tests to verify their correctness.

While most UI software falls into the *A* class *in its entirety*, large software is made from smaller parts, which in turn can be classified as either *S* or *A*. In the temperature display example above, emphasis is the problematic aspect in terms of precise formal specification—an *A* aspect. In contrast, the function `tooHot` solely operates in terms of the core domain logic, and is independent of its context or any human habits or practices—it belongs to class *S*.

This S/A separation also applies to Reacl's separation between a UI widget's event handler, the representation of the user's intention, and its dispatch in the `handle-message` function. All fall squarely within class A, which makes the separation less valuable than it might seem from a purely architectural perspective.

In order to bring down the costs of tests, we have to strive for a separation of *S* software parts from *A* software parts, and thus we want to write as little UI code as possible. In the remainder of this work we propose a discipline that supports this goal: we introduce reacl-c, a UI combinator library that is optimized for reusability and we make a case for the *functional view model*, a UI programming pattern.

12 From Reacl to reacl-c

Reacl allows for many different styles of component composition, which makes it hard to write widely applicable reusable abstractions. Over the years, patterns for Reacl components emerged, eventually leading to the creation of the *reacl-c* library [2]. The differences between Reacl and reacl-c focus on the relationship between components, and at what time the relationship is defined. Consider the call to `text-field` in the phonebook example: The reaction is part of the `text-editor` *class*, and thus effectively has to be the same for all components of that class—it's decided statically.

A developer might want to defer the decision on the reaction until run time, maybe because they want to use a component combinator that has to operate on state as well. In that case, they have to write a function that takes the `reacl/opt` value and `current-name` as arguments and build a combinator that works on these component constructor functions (instead of working on components), which is awkwardly indirect. To that end, reacl-c does away with the separation between classes and their components.

Another pattern that emerged from experience with Reacl is the relationship between parent and child, which is often best described by a lens [14]: A child component is often responsible for part of its parent's state. That means that the data flow from parent to child works like the `GET` operation of a suitable lens and the data flow from child to parent corresponds to the `PUTBACK` operation.

12.1 User-Interface Combinators

reacl-c inherits the Model-View-Update paradigm from Reacl. State is kept local to each item. The relationship between the states of two different items is mostly expressed with lenses. reacl-c's small set of flexible combinators allows developers to express abstractions that increase reuse and help sharply delineate the distinction of S and A code.

The unit of composition in reacl-c is called *item*. All reacl-c items have a notion of state, items may have a visual appearance, items may issue actions and messages, and items may perform controlled effects. For this overview it suffices to focus on (DOM) visuals and state. An item has the (imaginary) type `Item state`, where `state` is the type of the item's state.

Strings are primitive, state-agnostic items that simply display as themselves:

```
"Hello" : forall s. Item s
```

The `div` DOM combinator combines multiple items into one and allow adding additional styling. Given two items `i` and `j` of type `Item s`, the new item

```
(div {:style {:border "1px solid blue"}}
     i j) : Item s
```

displays `i` and `j` side-by-side (or on top of each other, depending on circumstances outside of the scope of this discussion) and draws a blue border around them. Optionally, these DOM combinators take special arguments that allow users to update state. The item

```
(button
 {:onClick (fn [x] (+ x 1))} "Inc") : Item Int
```

displays a button with the label “Inc” that increments its state when the user clicks. The handler function argument takes the current state as its first argument and returns a new state value for the next logical UI cycle.

focus is an item combinator that takes an item that operates on state of type s and a lens between types s and t and returns a new item that operates on type t. Given the increment button above as incButton, we can build a GUI for a tuple of two integers by combining focus and div:

```
(div
 (focus lens/first incBut)
 (focus lens/second incBut)) : Item (Int, Int)
```

All items above only update state but never display it. That's where the dynamic item constructor comes into play:

```
dynamic : forall s.(s -> Item s) -> Item s
```

This takes a function that gets the current state of type s and produces an item with that same state type. We can use dynamic to build an increment button that displays the current value as its label:

```
(c/dynamic
 (fn [i]
  (button {:onClick (fn [x] (+ x 1)} (str i)))))
```

The item combinator

```
isolate-state : forall s t.t -> Item t -> Item s
```

takes an initial value of type t and an item of type Item t and “runs” the item with the given initial state value. The resulting item can be used in any state context. This detaches an item's state from its surrounding state. A less drastic combinator is

```
local-state : forall s t.t -> Item (s, t) -> Item s
```

which takes an initial value of type t and an item that operates on state of type (s, t). The result is an item that operates on state of type s. From the perspective of the parent item, local-state introduces the right part of the state tuple to form the child's state. When viewed from the perspective of the child item, local-state hides the right part of the state tuple from the parent.

These combinators (along with a handful of others) enable programming techniques from pure functional programming to be carried over to GUI construction.

12.2 Phonebooks Redux

To further illustrate reacl-c, we re-use the phonebook domain model from section 12. We import the reacl-c namespaces along with active.data and active.clojure.

```
(ns phonebook
  (:require
    [reacl-c.core :as c :include-macros true]
    [reacl-c.dom :as dom :include-macros true]
    [reacl-c.main :as cmain]
    [active.clojure.lens :as lens]
   [active.data.record :refer-macros [def-record]]
    [active.data.realm :as realm]))
```

For our phonebook GUI we need a text-field, which we can build with a combination of c/dynamic and dom/input.

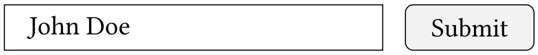
```
(def text-field
  (c/dynamic
   (fn [text]
    (dom/input
     {:value text
      :onChange (fn [old-text event]
                  (.-value (.-target event)))}))))
```

As shown above the :onChange handler describes how to update the current state. Here we see how reacl-c integrates with the native DOM bedrock of the browser: The :onChange handler takes a native DOM event object as an optional second parameter that contains the desired result text.

We can build a text editor with a Submit button, similar to the Reacl version above with the help of local-state. Here we can reuse text-field unmodified.

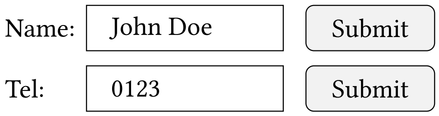
```
(def text-editor
  (c/dynamic
   (fn [text]
     (c/local-state
      text
      (dom/form
       {:onSubmit
         (fn [[outer-text current-text] event]
           (.preventDefault event)
           [current-text current-text])}
       (c/focus lens/second text-field)
      (dom/button {:type "submit"} "Submit"))))))
```

The local-state function pairs local state current-text with the item's state, called outer-text in the submit callback. (Thus, differently from Reacl, the local state becomes part of the state rather than being bound to a separate variable.) The call to focus that embeds text-field focusses the text-field component on that local state.


John Doe

Many relationships between different parts of GUI applications can be expressed as lenses, so the focus combinator makes for powerful glue. (Record field names defined by def-record from active.data.record are also lenses.) The following example shows a small UI that manages a single entry in our phonebook application.

```
(def entry-item
 (dom/div "Name:" (c/focus entry-name text-editor)
          "Tel:" (c/focus entry-phone-number
                          text-editor)))
```


Name: John Doe
Tel: 0123

We can evolve this single phonebook entry to an item that manages an entire phonebook (a sequence of entries) by using focus with a lens:

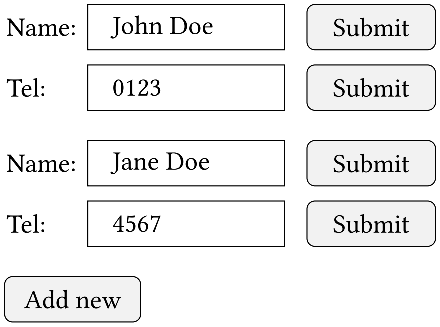
```
(def phonebook
  (c/dynamic
   (fn [entries]
     (apply
      dom/div
      (map (fn [idx]
             (c/focus (lens/at-index idx)
                      entry-item))
           (range (count entries)))))))
```

Here we map over the indices of the entries in the phonebook and use (c/focus (lens/at-index idx) ...) to focus on the entry item for each phonebook entry. This pattern is very common when dealing with lists, and we can distill it into a custom combinator:

```
(defn map-item [child-item]
  (c/dynamic
   (fn [xs]
     (apply
      dom/div
      (map (fn [idx]
             (c/focus (lens/at-index idx)
                      child-item))
           (range (count xs)))))))
(def phonebook-2 (map-item entry))
```

In contrast to Reacl components, items such as child-item are not wired to any other UI widgets, so the map-item combinator can use focus to establish a connection between its state and child-item's state. Here, we add a button that inserts a new entry into the phonebook, which works with the entire phonebook state.

```
(def empty-entry
  (entry entry-name "" entry-phone-number ""))
(def phonebook-with-add-button
  (dom/div
   phonebook-2
   (dom/button
    {:onClick
    (fn [phonebook _] (conj phonebook empty-entry))}
    "Add new")))
```


Name: John Doe
Tel: 0123
Name: Jane Doe
Tel: 4567

13 The View Model, Functionally

In the phonebook application above, when a user clicks the "Add new" button, the application inserts a new entry below the others. The user can then click on the name input field and enter the new entry's name. The newly inserted widget looks like any other list item, however. In order to give the user explicit feedback as to what happened on screen, we now want to highlight newly inserted entries. As we argued in Section 11, it is hard to formally specify what it means exactly for a UI widget to be emphasized. We should still extract the notion of emphasis into an isolated item combinator:

```
(defn emphasize [item]
  (dom/div {:style {:border "1px solid blue"}}
           item))
```

So far the domain model and the state of our items have been the same. Now we also have to keep track of which entry to emphasize. We therefore separate our domain model from the model of our UI state. This is a pattern well-known in the OO community as "Model-View-ViewModel" (MVVM) [23]. Our view model includes a phonebook entry from the core domain model, enriched with the information whether this particular entry should be emphasized. A phonebook view model consists of a sequence of such entries.

```
(def-record entry-vm
  [entry-vm-entry :- entry
   entry-vm-emphasized? :- realm/boolean])
```

It probably doesn't make sense to have more than one entry with entry-vm-emphasized? set to true. We provide a set of smart constructors and accessors that keep the constraint satisfied. First, we need an empty phonebook:

```
(def empty-phonebook [])
```

In our minimal phonebook example the only way to build larger phone books is to use an add-new-entry function:

```
(defn deemphasize [entry]
  (entry-vm-emphasized? entry false))
(defn add-new-entry [entries]
  (conj (mapv deemphasize entries)
        (entry-vm entry-vm-entry empty-entry
                  entry-vm-emphasized? true)))
```

add-new-entry is a pure function with precise semantics and thus a perfect candidate for testing.

These definitions constitute our view model. The corresponding UI code is straightforward. The new entry item entry-item-2 handles emphasis conditionally and then delegates on to the entry-item defined above.

```
(def entry-item-2
  (c/dynamic
   (fn [entry-vm]
     ((if (entry-vm-emphasized? entry-vm)
        emphasize identity)
      (c/focus entry-vm-entry entry-item)))))
(def phonebook-item-2 (map-item entry-item-2))
```

The phonebook still needs an "Add new" button. This button now simply calls `add-new-entry`:

```
(def phonebook-with-add-button-item-2
  (dom/div phonebook-item-2
           (dom/button {:onClick add-new-entry}
                       "Add new")))
```

The view-model pattern enabled by reacl-c allows moving part of the UI into S realm of the SPE classification. This yields the important architectural benefits of the Reacl model: no separate update code, modular components, no circular callbacks. Separating out the view model further enables testability in the parts of the UI code where it makes sense, and minimizes the amount of code where it does not.

14 Conclusion

UI programming and the architecture of UI program have evolved significantly since the MVC pattern, but the basic tenets of MVC are still in place. Object-oriented and functional approaches have had complementary trade-offs: Pure MVC featured good modularity from the start, but has struggled with keeping the view current with respect to the view. Conversely, the functional Model-View-Update pattern in its pure form solves the update problem but struggles with modularity. We have described the Reacl and reacl-c UI toolkits that address the modularity issues with Model-View-Update, and shown how to enhance Model-View-Update with functional view models to further modularize UI programs.

Acknowledgments

We thank David Frese, the principal designer of reacl-c, for assistance with the example code. We also thank the reviewers for their helpful comments.

References

[1] Active Group. 2014. Reacl – A ClojureScript library for programming with Facebook's React framework. https://github.com/active-group/reacl.

[2] Active Group. 2020. reacl-c - A web programming library for ClojureScript. https://github.com/active-group/reacl-c.

[3] Active Group. 2025. Active Data. https://github.com/active-group/active-data.

[4] Steven Brown. 1999. *Visual Basic 6 Complete.* John Wiley & Sons.

[5] Magnus Carlsson and Thomas Hallgren. 1993. Fudgets - A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 321–330. doi:10.1145/165180.165228

[6] Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. 2006. Evolution in software systems: foundations of the SPE classification scheme. *J. Softw. Maintenance Res. Pract.* 18, 1 (2006), 1–35. doi:10.1002/SMR.314

[7] Antony Courtney and Conal Elliott. 2001. Genuinely Functional User Interfaces. In *Proceedings of the Haskell Workshop* (Firenze, Italy).

[8] Evan Czaplicki. 2016. A Farewell to FRP. https://elm-lang.org/news/farewell-to-frp.

[9] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13).* ACM, New York, 411–422. doi:10.1145/2491956.2462161

[10] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97).* ACM, New York, 263–273. doi:10.1145/258948.258973

[11] Conal M. Elliott. 2007. Tangible functional programming. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07).* Association for Computing Machinery, New York, NY, USA, 59–70. doi:10.1145/1291151.1291163

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A functional I/O system or, fun for freshman kids. *SIGPLAN Not.* 44, 9 (Aug. 2009), 47–58. doi:10.1145/1631687.1596561

[13] Sigbjorn Finne and Simon Peyton Jones. 1996. Composing the user interface with Haggis. In *Advanced Functional Programming*, John Launchbury, Erik Meijer, and Tim Sheard (Eds.). Springer, Berlin, Heidelberg, 1–37.

[14] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es. doi:10.1145/1232420.1232424

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[16] Emden R. Gansner and John H. Reppy. 1993. *A multi-threaded higher-order user interface toolkit.* John Wiley & Sons, Inc., USA, 61–80.

[17] Halogen [n. d.]. https://purescript-halogen.github.io/purescript-halogen/. https://purescript-halogen.github.io/purescript-halogen/.

[18] Peter Henderson. 1982. Functional geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, Pennsylvania, USA) *(LFP '82).* ACM, New York, 179-–187. doi:10.1145/800068.802148

[19] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. doi:10.1145/361598.361623

[20] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016).* Association for Computing Machinery, New York, NY, USA, 33–44. doi:10.1145/2976002.2976010

[21] Trygve Reenskaug. [n. d.]. MVC – XEROX PARC 1978–79. https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html. Retrieved 2025-05.

[22] John H. Reppy. 1999. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, England.

[23] Josh Smith. 2009. Patterns - WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magagzine* 24, 02 (Feb. 2009).

[24] Edward Yourdon and Larry L. Constantine. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Prentice-Hall.

Received 2025-06-16; accepted 2025-07-21