Functional Shell and Reusable Components for Easy GUIs

D. Ben Knoble
Independent
Richmond, Virginia, USA
ben.knoble+funarch2023@gmail.com

Bogdan Popa
Independent
Cluj-Napoca, Cluj, Romania
bogdan@defn.io

arXiv:2308.16024v1 [cs.SE] 30 Aug 2023

Abstract

Some object-oriented GUI toolkits tangle state management with rendering. Functional shells and observable toolkits like GUI Easy simplify and promote the creation of reusable views by analogy to functional programming. We have successfully used GUI Easy on small and large GUI projects. We report on our experience constructing and using GUI Easy and derive from that experience several architectural patterns and principles for building functional programs out of imperative systems.

CCS Concepts: • **Software and its engineering** → **Publish-subscribe / event-based architectures**; **Reusability**; *Classes and objects*; *Extensible languages*.

Keywords: Reactive GUI, Functional wrapper

ACM Reference Format:
D. Ben Knoble and Bogdan Popa. 2023. Functional Shell and Reusable Components for Easy GUIs. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '23), September 8, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3609025.3609478

1 Introduction

Object-oriented programming is traditionally considered a good paradigm for building graphical (GUI) programs due to inheritance, composition, and specialization. Racket's GUI toolkit [12] is object-oriented, with message-passing widgets and mutable state. The Racket platform [14] provides the core class and object library for the GUI toolkit.

Figure 1 demonstrates typical Racket GUI code: it renders a counter with buttons to increment and decrement a number. First, we create a top-level window container, called a `frame%`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FUNARCH '23, September 8, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0297-6/23/09…$15.00
https://doi.org/10.1145/3609025.3609478

```
#lang racket/gui
(define f (new frame% [label "Counter"]))
(define container
  (new horizontal-panel% [parent f]))
(define count 0)
(define (update-count f)
  (set! count (f count))
  (define new-label (number->string count))
  (send count-label set-label new-label))
(define minus-button
  (new button% [parent container]
       [label "-"]
       [callback (λ _ (update-count sub1))]))
(define count-label
  (new message% [parent container]
       [label "0"]
       [auto-resize #t]))
(define plus-button
  (new button% [parent container]
       [label "+"]
       [callback (λ _ (update-count add1))]))
(send f show #t)
```

Figure 1. A counter GUI using Racket GUI's object-oriented widgets.

To lay out the controls horizontally, we nest a `horizontal-panel%` as a child of the window. We define the count state and a procedure to simultaneously update the count and its associated label. Next, we create the buttons and label for the counter. Lastly, we call the `show` method on the `frame%` to render it for the user.

The code in figure 1 has several shortcomings. It is verbose relative to the complexity of the GUI it describes and organized in a way that obscures the structure of the resulting interface. The programmer manually synchronizes application state, like the count, and UI state, like the message label, by mutation.

GUI Easy is a functional shell for Racket's GUI system based on observable values and function composition that aims to solve the problems with the imperative object-based APIs [26]. You can install the package `gui-easy` through

```
#lang racket/gui/easy
(define @count (@ 0))
(render
 (window
  #:title "Counter"
  (hpanel
   (button "-" (λ () (<~ @count sub1)))
   (text (~> @count number->string))
   (button "+" (λ () (<~ @count add1))))))
```

Figure 2. A counter GUI using GUI Easy's functional widgets.

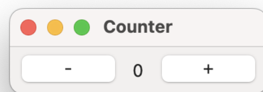

Figure 3. The rendered counter GUI on macOS.

the menu of DrRacket [11] or using the Racket command-line tool `raco` [27], after which you can run the examples in DrRacket or from your favorite programming environment.

With GUI Easy, the code in figure 2 resolves the previous shortcomings. As state, we define an observable `@count` whose initial value is the number `0`. Then, we `render` an interface composed of widgets like `window`, `hpanel`, `button`, and `text`. Widget properties, such as size or label, may be constant values or observables. The rendered widgets automatically update when their observable inputs change, similar to systems like React [29] and SwiftUI [1]. In this example, pressing the buttons causes the counter to be updated, which updates the text label.

In this report, we examine the difficulties of programming with object-oriented GUI systems and motivate the search for a different system in section 2, describe the main GUI Easy abstractions in section 3, report on our experience constructing large GUI programs in section 4, explore key architectural lessons in section 5, and explore related trends in GUI programming in section 6.

2 A Tale of Two Programmers

We present the origin stories for two projects. First, Bogdan describes his frustrations with Racket's GUI system that drove him to create GUI Easy. Second, Ben describes his desire to construct a large GUI program using a functional approach. The happy union of these two desires taught us the architectural lessons we present in section 4.

2.1 Quest for Easier GUIs

Bogdan's day job involved writing many small GUI tools for internal use. The Racket GUI framework proved an excellent way to build those types of tools as it provides fast iteration times, portability across major operating systems, and distribution of self-contained applications.

Over time, however, Bogdan was repeatedly annoyed by the same inconveniences. Racket's class system requires verbose code. Each project manages state in its own way. Racket GUI's primary means of constructing view hierarchies is to construct child widgets with references to their parent widgets, which makes composition especially frustrating since individual components must always be parameterized over their parent.

Since Racket GUI offers no special support for managing application state, Bogdan had to bring his own state management to the table, leading to ad hoc solutions for every new project. See `update-count` in figure 1 for an example of ad hoc state management. This motivated the observable abstraction in GUI Easy. In the next section, we will see how observables and observable-aware views combine to automatically connect GUI widgets and state changes.

Bogdan found it inconvenient that constructing most widgets requires a reference to a parent widget. Consider the following piece of Racket code:

```
(define f (new frame% [label "A window"]))
(define msg
  (new message% [parent f]
       [label "Hello World"]))
```

We cannot create the message object before the frame object in this case, since we need a `parent` for the message object. This constrains how we can organize code. To work around the issue, we can abstract over message object construction, but that needlessly complicates wiring up interfaces. This motivated Bogdan to come up with the view abstraction in GUI Easy. In section 3, we will see how views permit functional abstraction, enabling new organizational approaches that we will explore in section 4.

2.2 Embarking for the Town of Frosthaven

Ben enjoys boardgames with a group of friends, especially Frosthaven [4], the sequel to Gloomhaven. Due to its highly complex nature, Frosthaven includes lots of tokens, cards, and other physical pieces that the players must manipulate to play the game. This includes tracking monsters' health and conditions, the strength of six magical elements that power special abilities, and more. The original Gloomhaven game had a helper application for mobile devices to reduce physical manipulation; at one point, it appeared Frosthaven would not receive the same treatment.

Ben, a programmer, decided to solve the problem for his personal gaming group by creating his own helper application. But how? Having never created a complex GUI program,

```
#lang racket/gui/easy
(define (counter @count action)
  (hpanel
   (button "-" (λ () (action sub1)))
   (text (~> @count number->string))
   (button "+" (λ () (action add1)))))

(define @c1 (@ 0))
(define @c2 (@ 5))

(render
 (window
  #:title "Counters"
  (counter @c1 (λ (proc) (<~ @c1 proc)))
  (counter @c2 (λ (proc) (<~ @c2 proc)))))
```

Figure 4. Component re-use in GUI Easy. Multiple counter widgets can be created from a single definition.

Ben was intimidated by classic object-oriented systems like Racket's GUI toolkit. To a programmer with intimate knowledge of the class, method, and event relationships, such a system may feel natural. To the novice, GUI Easy represents a simpler, functional, path to interface programming.

GUI Easy makes it possible to build a complex system out of simple parts: functions and data. Ben was familiar with functional programming and grokked GUI Easy, so he started programming the Frosthaven Manager [20] with GUI Easy in 2022.

3 GUI Easy Overview

The goal of GUI Easy is to simplify user interface construction in Racket by wrapping its imperative API in a functional shell. GUI Easy can be broadly split up into two parts: *observables* and *views*.

Observables contain values and notify subscribed observers of changes to their contents. Section 3.1 explains the observable operators.

Views are representations of Racket GUI widget trees that, when rendered, produce concrete instances of those trees and handle the details of wiring state and widgets together. We discuss the view abstraction in more detail in section 3.2.

The core abstractions of observables and views correspond to a model-view-controller (MVC) architecture for graphical applications, as popularized by Smalltalk-80 [15, 21]. We describe the correspondence in section 3.3.

3.1 Observable Values

The core of the observable abstraction is that arbitrary observers can react to changes in the contents of an observable. Application developers programming with GUI Easy use a few core operators to construct and manipulate observables.

We create observables with `@`. By convention, we prefix observable bindings with the same sigil.

We can change the contents of an observable using `<~`. This procedure takes as arguments an observable and a procedure of one argument, representing the current value, to generate a new value. Every change is propagated to any observers registered at the time of the update.

We can derive new observables from existing ones using `~>`. This procedure takes an observable and a procedure of one argument, the current value. A derived observable changes with the observable it is derived from by applying its mapping procedure to the values of its input observable. In figure 4, the derived observable `(~> @count number->string)` changes every time `@count` is updated by `<~`; its value is the result of applying `number->string` to the value of `@count`. We cannot directly update derived observables.

We can peek at an observable with `obs-peek`, which returns the contents of the observable. This operation is useful to get point-in-time values out of observables when displaying modal dialogs or other views that require a snapshot of the state.

3.2 Views as Functions

Views are functions that return a `view<%>` instance, whose underlying details we will cover in section 5.2. Views might wrap a specific GUI widget, like a text message or button, or they might construct a tree of smaller views, forming a larger component. Both are synonymous with "view" in this report. We have already seen many examples of views like `text`, `hpanel`, and `counter`.

Views are typically observable-aware in ways that make sense for each individual view. For instance, the `text` view takes as input an observable string and the rendered text label updates with changes to that observable. Figure 4 shows an example of a reusable counter component made by composing views together.

Many Racket GUI widgets are already wrapped by GUI Easy, but programmers can implement the `view<%>` interface themselves in order to integrate arbitrary widgets, such as those from 3rd-party packages in the Racket ecosystem, into their projects.

3.3 Models, Views, and Controllers

The popular MVC architecture for graphical applications divides program modules into models of the application domain, views of the models, and controllers coupled with the views to translate user interactions into commands that affect the model [21].

Racket GUI applications can be organized according to the MVC architecture. In figure 1, the model is an integer `count`; the view is the combination of `button%` and `mesage%` objects, and the controller is the `update-count` procedure. Notice, however, that explictly grouping the view objects into a single reusable component requires contorting the code

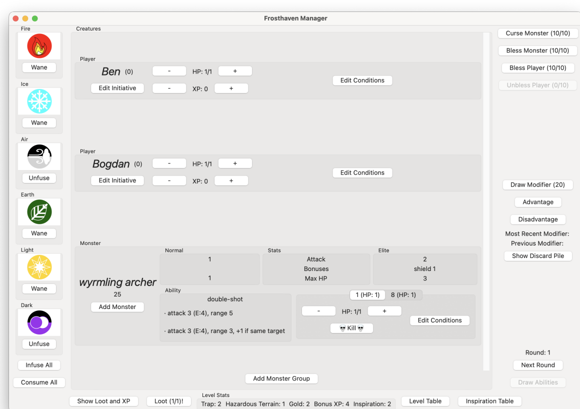

Figure 5. The Frosthaven Manager's main window on macOS.

responsible for object creation. There is no explicit support for the MVC pattern, though it can be used implicitly.

GUI Easy encourages an MVC-like architecture through the observable and view abstractions. Consider as an example figure 4: the observables `@c1` and `@c2` form the model, and each is distinguished from ordinary values. Similarly, the `counter` procedure is both a GUI Easy view and an MVC view. Finally, the controller's role is fulfilled by the `action` callback, which gives the `counter` consumer control over how user interactions are translated to model updates.

In summary, the MVC architecture encouraged by GUI Easy uses observables for models, GUI Easy views for views, and callbacks for controllers.

4 The Architecture of Frosthaven

In this section, we describe various pieces of a large GUI Easy application, the Frosthaven Manager.

At time of writing, the Frosthaven Manager includes approximately 5000 lines of Racket code. About half of that code makes up the main application by combining GUI Easy views with domain-specific code. Of the remaining lines, approximately 1000 implement the data structures and transformations responsible for the state of the game; 500 cover the images it draws; 750 implement three user-programmable data-definition languages[1]; 300 test the project; the remaining lines are small syntactic utilities. The Frosthaven Manager also has approximately 3000 lines of Scribble, a Racket prose and documentation language, which includes a how-to-play guide and developer reference.

The Frosthaven Manager manipulates many kinds of data. This includes game characters and their various attributes, monsters and their attributes, randomized loot, the status of elemental effects, and more. To organize and manipulate this data, Ben chose a "functional core, imperative shell" architecture [2].

The choice of a functional core and imperative shell has many benefits. For example, core code is independent of the choice of UI presentation and is independently testable or usable for other applications. Functional cores also simplify programmer reasoning about application data flow, keeping state change at the boundaries of the system.

In constructing the Frosthaven Manager, Ben organized the main data into immutable records, enumerations, and collections alongside pure functions that transform data according to the rules of the game. We thus say that the Frosthaven Manager uses a functional core.

Layered atop the functional core we find two more major components in the Frosthaven Manager: GUI-specific data and domain-specific views built on GUI Easy. In many ways, Ben took the functional approach here, too. GUI-related data is organized along typical idioms and paired with transformation functions. Despite these functional qualities, since most of the relevant data is observable or intended to be observable, the resulting system feels far more imperative. For example, pure transformations from the functional layers are paired with observable updates—akin to mutations—for real effect on the state of the GUI. As a result, though many important and reusable views seem pure, they are easily combined into a highly imperative system. These views and updates form the Frosthaven Manager's imperative shell.

The Frosthaven Manager's main GUI comprises many smaller reusable views. By analogy with functional programming's building blocks—functions—small reusable views permit us to construct large systems via composition. We will discuss the design principles behind reusable views in section 5.1.

5 Architectural Lessons

In this section, we cover the major lessons learned while developing these systems. First, reusable views (section 5.1) permit interface composition akin to functional composition by constraining how state is manipulated. Second, wrapping an imperative API with a functional shell (section 5.2) allows programmers to use functional techniques and architectures when constructing imperative systems. Third, inversion of control (section 5.3) creates an extensible application skeleton.

5.1 Reusable Views

Our experience building applications taught us to prefer reusable views where possible. Much like pure functions, a reusable view is composable and is subject to constraints on state manipulation. All the views provided by GUI Easy are reusable as described in this section.

[1] https://benknoble.github.io/frosthaven-manager/Programming_a_Scenario.html

There is one major design constraint on reusable views. *Views should not directly manipulate external state.* This is analogous to the rule for pure functions, and all the same arguments apply to show that manipulating external state makes a view less reusable. This leads naturally to the principle "data down, actions up," or *DDAU*. It also guides us to make decisions about which state to centralize at the highest levels of the GUI and which state to localize in reusable views.

DDAU prescribes how a reusable view should manipulate state. The "data down" prescription means that all necessary data must be inputs to a view. Recall the `counter` view from figure 4: the data needed to display the value of the counter was an observable input to the view called `@count`. The "actions up" prescription means that views should not directly manipulate state; instead, they should pass actionable data back to their caller, which is better positioned to decide how to manipulate state. Actions are represented by callbacks. For the `counter` view, the `action` callback is passed a procedure indicating whether the minus or plus button was clicked; the caller of the `counter` view decides how to react to user manipulations of the GUI.

It would be generally unsafe to mutate observable inputs, as they could be derived observables. Requiring informally that a particular view's observable inputs are not derived observables creates a trap for programmers that want to reuse the view in novel contexts and violates the principles of reusable views. Reusable views could take separate input and output observable formal arguments to work around this restriction, but that approach is generally less flexible and less convenient for the user than callbacks.

Callbacks are also easier to compose than separate input and output observables. For example, when a parent view uses a child view, it might specify the child's callback by wrapping the parent's own callback. The result is that events in the child are passed up through the parent, with the parent able to intercept, modify, and filter events from the child.

DDAU naturally bubbles application state up the layers of application architecture, so that the top-level of an application contains all of the necessary state. Callers pass the state, or a subset of it, down to various component views and provide procedures to respond to actions. This downward flow of state continues until we reach the bottom-most layer. Sometimes, however, we need state that is neither the caller's nor callee's responsibility. In such cases, a reusable view maintains local state which it is free to manipulate. This is in keeping with the tradition of optimizing functional programs by allowing interior—but invisible—mutability.

The benefits of reusable views are threefold. Small reusable views are amenable to independent testing. General-purpose views can be considered for extraction to a separate library, much like generic data-structure functions. Domain-specific views facilitate cohesion, such as visual style for a GUI application.

```
(define view<%>
  (interface ()
    [dependencies (->m (listof obs?))]
    [create (->m container/c widget/c)]
    [update (->m widget/c obs? any/c void?)]
    [destroy (->m widget/c void?)]))
```

Figure 6. The `view<%>` interface.

While reusable views are a GUI-specific idea, the notions of DDAU and constrained state management are also a more general lesson for functional programming: identifying patterns of state manipulation and constraining such state manipulation is a useful way to contain state in a smaller portion of code and to permit functional techniques in the remainder.

5.2 `view<%>`: Functional Shell, Imperative Core

The "Functional Core, Imperative Shell" architecture involves wrapping a core of pure functional code with a shell of imperative commands. In a twist on the paradigm, the core of GUI Easy views is an imperative object lifecycle, while its shell is functional. In this section, we describe that shell in detail and explain how it permits retaining functional programming techniques when dealing with imperative systems.

The GUI object lifecycle is embodied by the `view<%>` interface (figure 6). Implementations of the interface must know how to *create* widgets, how to *update* them in response to changed data dependencies, and how to *destroy* them if necessary [26]. They must also propagate data dependencies up the object tree. Data dependencies are any observable inputs to a view. The framework signals updates when dependencies change, allowing `view<%>`s to propagate updates to their wrapped widgets. Crucially, `view<%>` instances must be reusable, so they must carefully associate any internal state they need with each rendered widget.

To go from a `view<%>` to a functional view, all that remains is to wrap object construction in a function. Thus, the shell—the part that most library consumers interact with—is functional. Figure 7 shows an implementation of a custom `view<%>` and its function wrapper.

How does such a shell permit the use of functional programming techniques? We have already seen in the previous sections and in code examples that this shell abstracts away all the imperative details from most library consumers: until now, we have not needed to understand the imperative object-based API being wrapped in order to write GUI programs. Further, those GUI programs have used functional programming techniques, such as composition of reusable views. Even the Frosthaven Manager sticks mostly to GUI Easy's functional shell and is thus able to use the "Functional Core, Imperative Shell" architecture.

```
(require (prefix-in gui: racket/gui))
(define text%
  (class* object% (view<%>)
    (init-field @label) (super-new)
    (define/public (dependencies) (list @label))
    (define/public (create parent)
      (new gui:message% [parent parent]
           [label (obs-peek @label)]))
    (define/public (update widget what val)
      (send widget set-label val))
    (define/public (destroy widget) (void))))

(define (text @label)
  (new text% [@label @label]))
```

Figure 7. An implementation of a custom `view<%>` for displaying label text.

```
#lang racket/gui/easy
(require racket/class)
(define close! void)
(render
 (window
  #:title "Goodbye World"
  #:mixin (λ (window%)
            (class window% (super-new)
              (set! close!
                (λ ()
                  (when (send this can-close?)
                    (send this on-close)
                    (send this show #f))))))
  (button "Click Me!" (λ () (close!)))))
```

Figure 8. Using mixins to write a GUI Easy app whose window is closed when a button is clicked.

The key lesson for functional programmers here is that, when possible, wrapping an imperative API in a functional shell enables all the benefits of functional programming. For highly complex systems, like GUIs, to rewrite the entire system in a functional style may be impractical. Instead, it is more practical to reuse existing imperative or object-based work by wrapping it in a functional shell.

5.3 Inversion of Control

Inversion of control refers to an architecture wherein the main application provides procedures called by some framework, rather than by other application code. The framework is responsible for most of the coordinating activity, such as managing an event loop [19].

GUI Easy is one such framework. It manages the object lifecyle of `view<%>` instances, which is also the lifecyle of GUI Easy graphical applications. Calling `render` on a view kicks off that lifecycle, which is managed by the GUI Easy library and which calls into application code so that it may respond to user interaction.

Inversion of control leads to an extensible application skeleton: the backbone of the application is under the framework's control. Applications hang the meat of their tasks on the extension points provided by the framework. In the case of GUI Easy, those extension points are (a) the event handlers provided by standard components, also called the controllers in section 3.3, and (b) the `view<%>` interface for creating new framework-aware components. Being able to compose individual framework components into larger components also contributes to extensibility and reuse.

5.4 Challenges

Naturally, maintaining reusable components and programming against a functional shell is not without its challenges. What do you do when you need access to the underlying object-oriented API for a feature not exposed by existing wrappers? How do you handle a piece of nearly-global state whose usage is hard to predict when writing reusable components? Fortunately, both of these problems have solutions.

The problem of access to imperative behaviors is solved by GUI Easy conventions. In an object-oriented toolkit, we would subclass widgets as needed to create new behaviors. We cannot subclass a class we cannot access, for it is ostensibly hidden by the wrapper. In response, some GUI Easy views support a mixin [3, 5, 13, 16, 23] argument, a function from class to class. Mixins allows us to dynamically subclass widgets at runtime to override or augment their methods. Myers' "Goodbye World" program provides a good example: how can we include a button in the view that closes the window when such functionality is only present in the object-oriented toolkit? Figure 8 shows how: by using a mixin, we can get a reference to the window's `on-close` and `show` methods. The Frosthaven Manager uses mixins to implement window close behavior like in the "Goodbye World" program combined with a macro that implements the mixin-over-`(set! close! ...)` pattern; it also augments window close behavior so that closing the window can behave like accepting a choice. When mixins are insufficient, we can choose to write our own `view<%>` implementation to wrap any widget we desire. The Frosthaven Manager uses custom `view<%>`s to display rendered Markdown [17] files, for example.

The problem of global state is handled by functional programming techniques. Essentially, we have two choices: threading state or dynamic binding. If we are confident that the state will be required in all reusable views, we can thread the state as input from one view to the next, like threading a needle through all parts of the program. Threaded state is

```
(define (monster-group-view @monsters @env)
  (define @monster ...)
  (tabs @monsters
        (monster-view @monster @env)))

(define (monster-view @monster @env)
  (counter (monster->hp-text @monster @env)
           (λ (action) ...)))
```

Figure 9. Threading the `@env` argument from a view for monster groups to a view for a monster to a view for a monster's hit points.

the solution preferred by DDAU and reusable views. For example, the Frosthaven Manager threads an observable `@env` throughout the application so that simple arithmetic formulas with variables can be evaluated for monster information or scenario-specific attributes. As a result, many views take a `@env` argument, and many views pass a `@env` to child views. Figure 9 shows a simplified example.

Threading rarely-used state quickly becomes tedious and, when not needed everywhere, tangles unnecessary concerns. In response, we can use dynamic binding, which breaks some functional purity for convenience and allows us to refer to external state. Using dynamic binding makes views less reusable: they now have dependencies not defined by their inputs. Dynamic binding permits each view to only be concerned with the global state if absolutely necessary. The Frosthaven Manager threads state as much as possible but does use dynamic binding in rare instances. It is important to mention that using dynamic binding via Racket's parameters is not straightforward when working with the GUI system due to the multi-threaded environment and queued callbacks; to achieve dynamic-binding for the Frosthaven Manager, Ben had to both bind parameters in the GUI event threads and take care to spawn more event threads when new bindings were needed. This complexity may not be worth it in all applications.

6 Related Work

GUI Easy draws inspiration from Swift UI [1], another system that wraps an imperative GUI framework in a functional shell. Other sources of inspiration include Clojure's Reagent [28] and JavaScript's React [29].

The Elm [10] programming language strictly constrains component composition to the data down, actions up style. Clojure's re-frame [31] library builds on Reagent [28] to add more sophisticated state management. This includes a global store and effect handler, akin to GUI Easy's observables and update procedures, and queries, akin to GUI Easy's derived observables.

Frappé [9] is an implementation of FRP in Java that wraps an imperative API (Java Beans) in a declarative shell. Both GUI Easy and Frappé implement a "push" model for propagation of values through the dependency graph: *behaviors* in Frappé hold values and support the registration of listeners to be notified when their held values change, like observables in GUI Easy. Unlike Frappé, GUI Easy does not have an explicit notion of *events*. Instead, observables may be directly updated in response to callbacks.

In Racket, FrTime [6] implements a push-based FRP language for GUIs and other tasks. The FrTime language extends a subset of the Racket language to make signal values first-class. By contrast, GUI Easy is a regular library built on top of the Racket language—a conscious choice in order to make it straightforward to bring GUI Easy into existing Racket programs. One symptom of this choice is that, while FrTime signals can be displayed in a continuously variable manner with support from editors like DrRacket, GUI Easy observables are regular Racket values and are displayed as such. FrTime and GUI Easy both track state by using mutation internally and both FrTime *behaviors* and GUI Easy observables get updated asynchronously in response to changes.

Fred [18] is FrTime's wrapper around Racket GUI. It wraps the object-oriented API of Racket GUI by subclassing Racket GUI widgets to work with FrTime signal values. By contrast, GUI Easy views are separate classes that implement the `view<%>` interface. Despite this difference, both frameworks perform similar operations in order to connect their reactive abstractions to the underlying widgets. FrTime makes use of macros to generate most of its wrapper code, whereas GUI Easy views are implemented manually. Unlike GUI Easy, Fred does not hide the details of the Racket class system from the end user. Because its widgets sublass Racket GUI widgets, it has the same order-of-definition constraints as Racket GUI that we described in section 2.1.

Flapjax [22] is a push-based FRP implementation. It provides both a compiler from the Flapjax language to JavaScript and a standalone library. Similar to FrTime for Racket, the Flapjax language extends JavaScript to make behaviors first-class, implicitly lifting expressions to work over behaviors where necessary. The Flapjax compiler is optional and the standalone library can be used directly from JavaScript without compiler support, like GUI Easy can be from Racket. While GUI Easy observables get updated asynchronously and independently, updates in Flapjax are propagated through the dependency graph in topological order, avoiding potential inconsistencies between behaviors that share part of the dependency graph.

The Andrew toolkit and Garnet system, among others of that time, knew that the MVC architecture tightly couples views and controllers [24, 25]. Typical solutions involve not separating views and controllers or dropping the controller altogether [24]. DDAU from section 5.1 encourages decoupling the view and controller by the use of callbacks: they

provide the same interposition points a typical controller would use to respond to user interaction, and they provide different view instances the ability to respond with different model updates. This is especially important when the view can display many different models. Decoupling views and controllers also allow combining controllers when combining views. In the Garnet system, however, "spaghetti" callbacks are avoided by providing a small set of flexible Interactors and by using formulated constraints to tie together interactions and updates [24].

Inversion of control has a long history: the development of the Tajo [32] and Mesa [30] systems called it the "Hollywood Principle." Myers, developing the Garnet system, similarly separated monitoring processes from the users application code, providing hooks for the application to respond to events from the framework [24].

In the language of Johnson and Foote [19], we ask whether GUI Easy is a "white-box" or "black-box" inversion-of-control framework. White-box frameworks, so-called because they are transparent, typically require programs to subclass and add methods to framework components, which requires understanding implementation details. In contrast, black-box frameworks are an "evolutionary goal," in which opaque components communicate only via a shared protocol. Black-box frameworks are thus easier to learn and use. White-box frameworks typically maintain global state, while black-box frameworks see state shared explicitly when needed. Given these criteria, we can confidently state that the object-oriented Racket GUI toolkit is a white-box framework. GUI Easy is principally black-box, relying on a protocol of observables for state and procedures for communication. Yet GUI Easy provides escape hatches of varying complexity that bring back the expert flavor of white-box frameworks; namely, mixins and the `view<%>` interface. To give credit where it is due, we recognize that abstracting is much easier given a plethora of worked examples—we would not have the experience to develop GUI Easy without Racket's GUI toolkit. On the flipside, using GUI Easy has proven to be a good way to learn how to use Racket's GUI toolkit!

7 Conclusion

We have reported on the difficulties of programming stateful GUIs with imperative, object-based APIs. We also described a functional wrapper around Racket's object-oriented GUI library that aims to solve some of those shortcomings. GUI Easy has been successfully used for small and large projects, including the Frosthaven Manager discussed in this report. We derived several architectural principles from the construction of both projects: functional shells over imperative APIs enable programmers to use functional programming techniques even when dealing with a system whose underlying implementation is imperative. Extensible hooks are necessary in functional shells to permit access to the underlying systems where needed. Reusable components, much like pure functions, should not mutate external state. Like pure functions, reusable components are independently testable and are easily composed with one another.

Acknowledgments

Ben is grateful to Savannah Knoble, Derrick Franklin, John Hines, and Jake Hicks for playtesting the Frosthaven Manager throughout development, and to Isaac Childres for bringing us the wonderful world of Frosthaven.

We thank the anonymous reviewers, our shepherd Shriram Krishnamurthi, and our early readers Jeff Terrell, Marc Kaufmann, Matthew Flatt, and Robby Findler for their insightful comments.

References

[1] Apple. SwiftUI. 2023. https://developer.apple.com/xcode/swiftui/ Retrieved June 2023.

[2] Gary Bernhardt. Functional Core, Imperative Shell. 2012. https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell

[3] Gilad Bracha. The Programming Languages Jigsaw: Mixins, Modularity, and Inheritance. PhD dissertation, University of Utah, 1992. https://bracha.org/jigsaw.pdf Ch. 3

[4] Cephalofair Games. Frosthaven. 2023. https://cephalofair.com/pages/frosthaven

[5] William R. Cook. A Denotational Semantics of Inheritance. PhD dissertation, Brown University, 1989. https://www.cs.utexas.edu/~wcook/papers/thesis/cook89.pdf Ch. 10

[6] Gregory Cooper and Shriram Krishnamurthi. FrTime: Functional Reactive Programming in PLT Scheme. Brown, CS-03-20, 2004. https://cs.brown.edu/research/pubs/techreports/reports/CS-03-20.html

[7] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proc. 15th European Conference on Programming Languages and Systems*, ESOP'06, pp. 294–308, 2006. doi:10.1007/11693024_20

[8] Gregory Harold Cooper. Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language. PhD dissertation, Brown University, 2008. https://cs.brown.edu/people/ghcooper/thesis.pdf

[9] Antony Courtney. Frappé: Functional Reactive Programming in Java. In *Proc. Practical Aspects of Declarative Languages*, 2001. https://doi.org/10.1007/3-540-45241-9_3

[10] Evan Czaplicki. Elm - delightful language for reliable web applications. 2021. https://elm-lang.org Retrieved June 2023.

[11] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming* 12(2), pp. 159–182, 2002.

[12] Matthew Flatt, Robert Bruce Findler, and John Clements. GUI: Racket Graphics Toolkit. PLT Design Inc., PLT-TR-2010-3, 2010. https://racket-lang.org/tr3/

[13] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pp. 171–183, 1998. doi:https://doi.org/10.1145/268946.268961

[14] Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. https://racket-lang.org/tr1/

[15] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley Publishers., 1983.

[16] David Goldberg, Robert Bruce Findler, and Matthew Flatt. Super and Inner—Together at Last! In *Proc. Object-Oriented Programming, Languages, Systems, and Applications*, 2004. http://www.cs.utah.edu/plt/publications/oopsla04-gff.pdf

[17] John Gruber. Daring Fireball: Markdown. 2023. https://daringfireball.net/projects/markdown/ Retrieved June 2023.

[18] Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages. In *Proc. Functional and Logic Programming*, FLOPS 2006, 2006. https://link.springer.com/chapter/10.1007/11737414_18

[19] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming* 1(2), pp. 22–35, 1988. http://www.laputan.org/drc/drc.html

[20] D. Ben Knoble. frosthaven-manager. 2022. https://github.com/benknoble/frosthaven-manager

[21] Glenn E. Krasner and Stephen T. Pope. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of Object-Oriented Programming* 1(3), pp. 26–49, 1988. https://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf

[22] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proc. ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA 2009, 2009. https://dl.acm.org/doi/10.1145/1640089.1640091

[23] David A. Moon. Object-oriented programming with Flavors. In *Proc. ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 1–8, 1986. https://www.cs.tufts.edu/comp/150FP/archive/david-moon/flavors.pdf

[24] Brad A. Myers. A New Model for Handling Input. *ACM Transactions on Information Systems* 8(3), pp. 289–320, 1990. doi:https://doi.org/10.1145/98188.98204

[25] Andrew J. Palay, Wilfred J. Hansen, Michael L. Kazar, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters. The Andrew Toolkit—An Overview. In *Proc. USENIX Winter Conference*, pp. 9–22, 1988.

[26] Bogdan Popa. Announcing GUI Easy. 2021. https://defn.io/2021/08/01/ann-gui-easy/

[27] raco: Racket Command-Line Tools. 2010. https://docs.racket-lang.org/raco/index.html

[28] reagent-project. Reagent. 2023. https://github.com/reagent-project/reagent Retrieved June 2023.

[29] Meta Open Source. React. 2023. https://react.dev Retrieved June 2023.

[30] Richard E. Sweet. The Mesa Programming Environment. *ACM SIGPLAN Notices* 20(7), pp. 216–229, 1985. doi:https://doi.org/10.1145/17919.806843

[31] Day 8 Technology. re-frame. 2023. https://github.com/day8/re-frame Retrieved June 2023.

[32] Donald C. Wallace. Tajo Functional Specification Version 6.0, Xerox Internal Document, 1980.

Received 2023-06-01; accepted 2023-06-28