DYNAMICS
OF CHANGE:

Why Reactivity Matters

**TAME THE
DYNAMICS OF
CHANGE BY
CENTRALIZING
EACH CONCERN IN
ITS OWN MODULE**

ANDRE MEDEIROS

Professional programming is about dealing with software at scale. Everything is trivial when the problem is small and contained: it can be elegantly solved with imperative programming or functional programming or any other paradigm. Real-world challenges arise when programmers have to deal with large amounts of data, network requests, or intertwined entities, as in UI (user interface) programming.

Of these different types of challenges, managing the dynamics of change in a code base is a common one that may be encountered in either UI programming or the back end. How to structure the flow of control and concurrency among multiple parties that need to update one another with new information is referred to as *managing change*. In both UI programs and servers, concurrency is typically present and is responsible for most of the challenges and complexity.

Some complexity is accidental and can be removed. Managing concurrent complexity becomes difficult when the amount of essential complexity is large. In those cases, the interrelation between the entities is complex—and cannot be made less so. For example, the requirements themselves may already represent essential complexity. In an online text editor, the requirements alone may determine that a keyboard input needs to change the view, update text formatting, perhaps also change the table of contents, word count and paragraph count, request the document to be saved, and take other actions.

Because essential complexity cannot be eliminated, the alternative is to make it as understandable as possible, which leads to making it maintainable. When it comes to complexity of change around some entity Foo, you want to understand what Foo changes, what can change Foo, and which part is responsible for the change.

HOW CHANGE PROPAGATES FROM ONE MODULE TO ANOTHER

Figure 1 is a data flow chart for a code base of e-commerce software, where rectangles represent modules and arrows represent communication. These modules are interconnected as requirements, not as architectural decisions. Each module may be an object, an object-oriented class, an actor, or perhaps a thread, depending on the programming language and framework used.

An arrow from the `Cart` module to the `Invoice` module (figure 2a) means that the cart changes or affects the state in the invoice in a meaningful way. A practical example of this situation is a feature that recalculates the

Managing concurrent complexity becomes difficult when the amount of essential complexity is large.

FIGURE 1: DATA FLOW FOR A CODEBASE OF AN E-COMMERCE SOFTWARE

total invoicing amount whenever a new product is added to the cart (figure 2b).

The arrow starts in the `Cart` and ends in the `Invoice` because an operation internal to the `Cart` may cause the state of the `Invoice` to change. The arrow represents the dynamics of change between the `Cart` and the `Invoice`.

Assuming that all code lives in some module, the arrow

FIGURE 2: THE CART CHANGES THE INVOICE

cannot live in the space between; it must live in a module, too. Is the arrow defined in the `Cart` or in the `Invoice`? It is up to the programmer to decide that.

Passive Programming

It is common to place the arrow definition in the arrow tail: the `Cart`. Code in the `Cart` that handles the addition of a new product is typically responsible for triggering the `Invoice` to update its invoicing data, as demonstrated in the chart and the Kotlin (<https://kotlinlang.org/>) code snippet in figure 3.

The `Cart` assumes a proactive role, and the `Invoice`

FIGURE 3: **PASSIVE PROGRAMMING WITH CODE IN TAIL**

takes a passive role. While the `Cart` is responsible for the change and keeping the `Invoice` state up to date, the `Invoice` has no code indicating that the update is coming from the `Cart`. Instead, it must expose `updateInvoicing` as a public method. On the other hand, the cart has no access restrictions; it is free to choose whether the `ProductAdded` event should be private or public.

Let's call this programming style *passive programming*, characterized by remote imperative changes and delegated responsibility over state management.

Reactive Programming

The other way of defining the arrow's ownership is *reactive programming*, where the arrow is defined at the arrow head: the `Invoice`, as shown in figure 4. In this setting, the `Invoice` listens to a `ProductAdded` event happening in the cart and determines that it should change its own internal invoicing state.

The `Cart` now assumes a broadcasting role, and the `Invoice` takes a reactive role. The `Cart`'s responsibility is to carry out its management of purchased products, while providing notification that a product has been added or removed.

Therefore, the `Cart` has no code that explicitly indicates that its events may affect the state in the `Invoice`. On the other hand, the `Invoice` is responsible for keeping its own invoicing state up to date and has the `Cart` as a dependency.

The responsibilities are now inverted, and the `Invoice` may choose to have its `updateInvoicing` method private or public, but the `Cart` must make the `ProductAdded`

FIGURE 4: REACTIVE PROGRAMMING WITH CODE IN HEAD

event public. Figure 5 illustrates this duality.

The term *reactive* was vaguely defined in 1989 by Gérard Berry.¹ The definition given here is broad enough to cover existing notions of reactive systems such as spreadsheets, the actor model, Reactive Extensions [Rx], event streams, and others.

FIGURE 5: PUBLIC VS. PRIVATE

PROGRAMMING	"PRODUCT ADDED" EVENT IN THE CART	UPDATE INVOICING DATA METHOD IN THE INVOICE
Passive	private or public	public
Reactive	public	private or public

PASSIVE VERSUS REACTIVE FOR MANAGING ESSENTIAL COMPLEXITY

In the network of modules and arrows for communication of change, where should the arrows be defined? When should reactive programming be used and when is the passive pattern more suitable?

There are usually two questions to ask when trying to understand a complex network of modules:

- ➔ Which modules does module X change?
- ➔ Which modules can change module X?

The answers depend on which approach is used: reactive or passive, or both. Let's assume, for simplicity, that whichever approach is chosen, it is applied uniformly across the architecture.

For example, consider the network of e-commerce modules shown in figure 6, where the passive pattern is used everywhere.

To answer the first question for the `Invoice` module (Which modules does the invoice change?), you need only to look at the code in the `Invoice` module, because it owns the arrows and defines how other modules are remotely changed from within the `Invoice` as a proactive component.

FIGURE 6: **FREQUENT PASSIVE PATTERN**

To discover which modules can change the state of the `Invoice`, however, you need to look for all the usages of public methods of the `Invoice` throughout the code base.

In practice, this becomes hard to maintain when multiple other modules may change the `Invoice`, which is the case in essentially complex software. It may lead to situations where the programmer has to build a mental model of how multiple modules concurrently modify a piece of state in the module in question. The opposite alternative is to apply the reactive pattern everywhere, illustrated in figure 7.

To discover which modules can change the state of the `Invoice`, you can just look at the code in the `Invoice` module, because it contains all “arrows” that define dependencies and dynamics of change. Building the mental model of concurrent changes is easier when all relevant entities are co-located.

On the other hand, the dual concern of discovering which other modules the `Invoice` affects can be answered only by searching for all usages of the `Invoice` module’s public broadcast events.

FIGURE 7: **FREQUENT REACTIVE PATTERN**

When arranged in a table, as in figure 8, these described properties for passive and reactive are dual to each other.

The pattern you choose depends on which of these two questions is more commonly on a programmer’s mind when dealing with a specific code base. Then you can pick the pattern whose answer to the most common question is “look inside,” because you want to be able to find the answer quickly. A centralized answer is better than a distributed one.

While both questions are important in an average code base, a more common need may be knowing how a particular module works. This is why reactivity matters: you usually need to know how a module works before looking at what the module affects.

Because a passive-only approach generates *irresponsible* modules (they delegate their state management to other modules), a reactive-only approach is a more sensible default choice. That said, the passive pattern is suitable for data structures and for creating a hierarchy of ownership. Any common data structure (such as a hash map) in object-oriented programming is a passive module, because it exposes methods that allow changing its internal state. Because it delegates the responsibility of answering the question “When does it change?” to whichever module contains the data-structure object, it

FIGURE 8: DUAL PROPERTIES

	PASSIVE	REACTIVE
How does it work?	Find usages	Look inside
What does it affect?	Look inside	Find usages

creates a hierarchy: the containing module as the parent and the data structure as the child.

MANAGING DEPENDENCIES AND OWNERSHIP

With the reactive-only approach, every module must statically define its dependencies to other modules. In the `Cart` and `Invoice` example, `Invoice` would need to statically import `Cart`. Because this applies everywhere, all modules would have to be singletons. In fact, Kotlin's `object` keyword is used (in Scala as well) to create singletons.

In the reactive example in figure 9, there are two concerns regarding dependencies:

FIGURE 9: **REACTIVE-ONLY APPROACH**

```
package my.project

import my.project.Cart // This is a singleton

public object Invoice { // This is a singleton too
    fun updateInvoicing(product: Product) {
        // ...
    }

    fun setup() {
        Cart.onProductAdded { product ->
            this.updateInvoicing(product)
        }
    }
}
```

- ➔ What the dependency is: defined by the `import` statement.

- ➔ How to depend: defined by the event listener.

The problem with singletons as dependencies relates only to the *what* concern in the reactive pattern. You would still like to keep the reactive style of *how* dependencies are put together, because it appropriately answers the question, “How does the module work?”

While reactive, the module being changed is statically aware of its dependencies through imports; while passive, the module being changed is unaware of its dependencies.

So far, this article has analyzed the passive-only and reactive-only approaches, but in between lies the opportunity for mixing both paradigms: keeping only the *how* benefit from reactive, while using passive programming to implement the *what* concern.

The `Invoice` module can be made passive with regard to its dependencies: it exposes a public method to allow another module to set or inject a dependency. Simultaneously, `Invoice` can be made reactive with regard to how it works. This is shown in the example code in figure 10, which yields a hybrid passively reactive solution:

- ➔ How does it work? Look inside (reactive).

- ➔ What does it depend on? Injected via a public method (passive).

This would help make modules more reusable, because they are not singletons anymore. Let’s look at another example where a typical passive setting is converted to a passively reactive one.

FIGURE 10: A HYBRID PASSIVELY REACTIVE SOLUTION

```
package my.project

public object Invoice {
    fun updateInvoicing(product: Product) {
        // ...
    }

    private var cart: Cart? = null

    public fun setCart(cart: Cart) {
        this.cart = cart
        cart.onProductAdded { product ->
            this.updateInvoicing(product)
        }
    }
}
```

EXAMPLE: ANALYTICS EVENTS

It is common to write the code for a UI program in passive-only style, where each different screen or page of the program uses the public methods of an `Analytics` module to send events to an `Analytics` back end. The example code in figure 11 illustrates this.

The problem with building a passive-only solution for analytics events is that every single page needs to have code related to analytics. Also, to understand the behavior of analytics, you must study it scattered throughout the code. It's desirable to separate the analytics aspect from the core features and business logic concerning a page

FIGURE 11: **PASSIVE-ONLY APPROACH**

```
// In the LoginPage module
package my.project

import my.project.Analytics

val loginButton = //...
loginButton.addClickListener { clickEvent ->
    Analytics.sendEvent('User clicked the login button')
}
```

such as the `LoginPage`. Aspect-oriented programming² is one attempt at solving this, but it is also possible to separate aspects through reactive programming with events.

In order to make the code base reactive only, the `Analytics` module would need to statically depend on all the pages in the program. Instead, you can use the passively reactive solution to make the `Analytics` module receive its dependencies through a public injection method. This way, a parent module that controls routing of pages

can also bootstrap the analytics with information on those pages. See the example in figure 12.

MIND THE ARROWS

Introducing reactive patterns in an architecture can help better define which module *owns* a relationship of change between two modules. Software architectures

FIGURE 12: **PUBLIC INJECTION METHOD**

```
// In the Analytics module
package my.project

public object Analytics {
  public fun inject(loginPage: Page) {
    loginPage.loginButton.addClickListener { clickEvent ->
      this.sendEvent('User clicked the login button')
    }
  }

  private fun sendEvent(eventMessage: String) {
    // ...
  }
}
```

for essential complex requirements are often about structuring the code in modules, but do not forget that the arrows between modules also live in modules. Some degree of reactivity matters because it creates separation of concerns. A particular module should be responsible for its own state. This is easily achievable in an event-driven architecture, where modules do not invasively change each other. Tame the dynamics of change by centralizing each concern in its own module.

References

1. Berry, G. 1989. Real time programming: special purpose or general purpose languages. [Research Report] RR-1065. INRIA (French Institute for Research in Computer Science and Automation); <https://hal.inria.fr/inria-00075494/document>.
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J. 1997. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*: 220–242.

Andre Medeiros is a web and mobile developer at Futurice. He is known for his involvement with reactive programming for user interfaces, particularly with the *ReactiveX* libraries. Medeiros has built JavaScript libraries and tools such as *Cycle.js* and *RxMarbles*. He has an M.Sc. in theoretical computer science.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.