A survey on Deprecating the Observer Pattern

JOEL BARTELHEIMER, Technische Hochschule Mittelhessen–University of Applied Sciences, Germany

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Additional Key Words and Phrases: design patter, observer pattern, event handling, data-flow language, reactive programming, user interface programming, scala

1 INTRODUCTION

In contrast to traditional batch mode programs modern applications are reactive and event driven. examples like the GUI reactive is hard and errorprone, dealing with continuous event occurrence and user input requires a considerable amount of engineering.

quote an Adobe presentation from 2007 [Parent 2007] on current production systems:

- 1/3 of the code in Adobes desktop applications is devoted to event handling logic
- 1/2 of the bugs reported during a product cycle exist in this code

A programming paradigm well suited for these event-driven and interative applications is reactive programming reactive programming gained popularity

early approach in reactive programming is the use of the observer Pattern

1.1 UI Event handling

in the paper "Deprecating the Observer Pattern" the authors criticize the use of the observer pattern for reactive programs, the title already is a hard claim by itself.

2 OBSERVER PATTERN

The observer pattern was originally published in [Gamma et al. 1995]. It is one of the behavior patterns by the Gang of Four. The intent of the observer pattern is a one-to-many dependency with an automatic change propagation. When the state of an object (one) changes, all it dependents objects (many) are notified automatically.

Author's address: Joel Bartelheimer, Technische Hochschule Mittelhessen–University of Applied Sciences, Giessen, Hessen, 35390, Germany, joel.bartelheimer@mni.thm.de.

2018. 0360-0300/2018/4-ART2 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

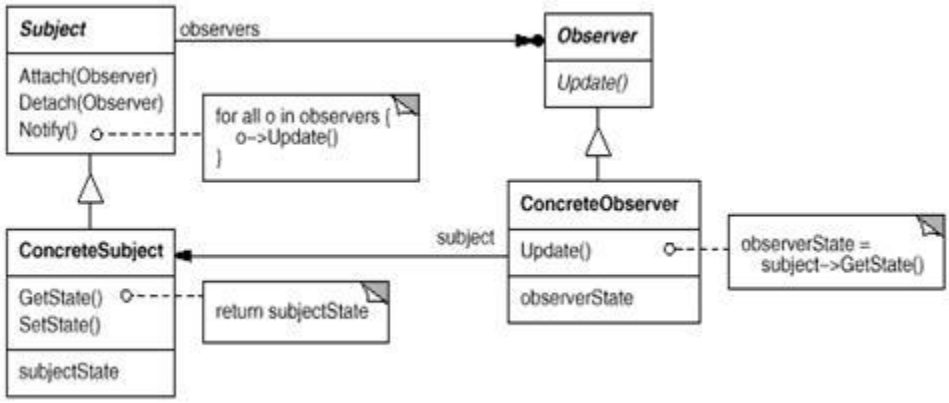

Fig. 1. Structure of the observer pattern [Gamma et al. 1995]

The observer pattern consists of four components, their relations and methods are shown in figure 1.

Subject Provides an interface for installing (*attach*) and uninstall (*detach*) observers. Moreover, the subject knows its observers and the number of observers is not limited.

Observer The *update* method is the observers updating interface for receiving changes in a subject.

ConcreteSubject Stores the state, where the observers are interested in and also initiates the notification when the state changes.

ConcreteObserver Implements an updating interface to keep its stored state of the subject consistent.

In the original source various slightly different implementation are discussed. One of them deals with the update and how the observers receive the changed data. Often the subject passes additional information as an argument to the *update* method, e.g. a changed mouse position. To avoid observer-specific update protocols, two extreme cases – *push model* and *pull model*– are described. With the *push model*, the subject sends detailed information to the observers about the changed state, even if the observers are not interested. On the opposite, nothing is sent by the subject only the change notification, the observers have to ask for the information afterward [Gamma et al. 1995]. Usually implementations, e.g. the Java *Observable* implementation [Oracle 2017], support both models.

For event handling, the subject receives the actual event, e.g. a mouse click, and notifies all its observers. And because of the Observer pattern, the event consumers are decouple from the source of the event. Therefore, typical use cases for the observer pattern are graphical user interfaces. Also many GUI related frameworks closely follow the observer pattern, e.g. Smalltalk's Model/View/Controller or Java Swing [Gamma et al. 1995; Maier et al. 2010].

2.1 Violations of software engineering principles

In [Maier and Odersky 2012] it is shown by means of code exmples that the observer pattern violates a set of software engineering principles, particularly for the use of event handling in interactive GUI applications. The violated principles are:

Side-effects Often several observers are used to implement a single Operation, e.g. a `onClick` and a `onRelease` observer are necessary to implement a drag and drop

operation. Therefore, the observer patterns promote side-effects and already on the API level.

Encapsulation In many cases multiple observers are used to simulate a state machine. The state, e.g. a dragged object, is stored in a variables within a shared scope. As this broader scope of these shared variables escapes the scope of the observers, the observer pattern violates the encapsulation principle.

Composability Observers are registered during runtime on various installation points. Even if multiple observers deal with a single concern, like a drag and drop operation, it is not possible to, e.g. easily add or remove drag behavior.

Resource management The installation and uninstallation of a observer has to be managed explicitly. The mouse movement for instance, should only be observed during a drag operation, due to performane. Therefor the client is in charge to manage the observers life-time.

Separation of concerns Usually a observer deals with more than on concern. For instance, recalculating and refreshing the state and also drawing the new state changes. That a single observer deals with multiple concerns is not recommendable, the concerns shall be seperated.

Data consistency The separation of concerns can be archived by limiting the concerns a observer deals with to one and then the observer itself publishes events on change. Each concern can then observe its dependent concerns. But unfortunately, there is no guarantee for data consistency when a observer listens to multiple changes.

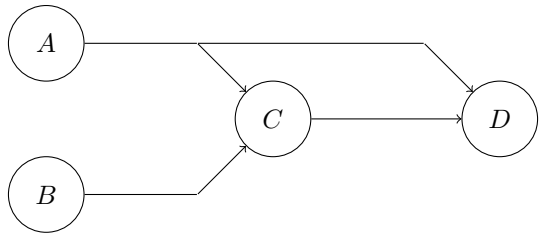

Fig. 2. Dependency graph

Figure 2 shows a dependency graph where C observes changes from A and B and D observes the resulting C and also A. When A emits a change event, D can not determine if C has already consumed the change or not. The short moment, where C has not consumed the change event, is called a *glitch*.

Uniformity Every subject provides its own installation point for observers, which decreases code uniformity.

Abstraction The use of heavyweight interfaces is promoted by the observer pattern. Often, much more methods than just one to install a specific observer is provided. Therefore, it is not possible to abstract over a precise event source, e.g. when we want to use pointer events instead of mouse events.

Semantic distance The observer pattern reverses the control flow – the subject invokes the `update()` method of the observer – which results in more boilerplate code and makes the code harder to understand.

Nearly half of the violations are caused by dealing with multiple observers. But exactly the feature of chaining and composing multiple observers together is not mentioned in the original source [Gamma et al. 1995].

3 REACTIVE PROGRAMMING

Reactive programming is a paradigm which provides dedicated programming abstraction for interactive and event-driven applications. As mentioned in the introduction, designing, implementing and maintaining reactive application is not an easy task. The code of a program is triggered asynchronously by the occurrence of events. For this reason, developers struggle to understand the control flow of an application. The approach of reactive programming is, to express these complex reactive behaviors in an intuitive and declarative way. The following are the core principles of reactive programming [Salvaneschi et al. 2015, 2017]:

Declarativeness Instead of computational steps, to derive a new component state based on a prior state, programmers address *how* components functionally depend on each other.

Abstraction over change propagation Change Propagation of dependent values is done implicit by the underlying execution model instead of manually by the developer.

Composability Reactive computations can be composed to more complex computations by using provided abstractions (e.g., combinators/operator for event streams).

Favoring data flow over control flow Normally the excecution of an application follows the control flow. In reactive programming the computation is driven by events and data and how they flow through the system.

To explain these principle a short example is given:

```
var1 = 1
var2 = 2
var3 = var1 + var2
```

In conventional sequential imperative programming, the variavle `var3` has the initial value of the sum of `var1` and `var2`, which is 3. And when afterwards a new value is assigned to variable `var1` or `var2` the value of `var3` remains. Only with an explicit assignment, the value of varialbe `var3` can be changed. In contrast to reactive programming, where variable `var3` is always up-to-date. Whenever the value of `var1` or `var2` changes, the reactive execution model recomputes the new value for `var3` [Bainomugisha et al. 2013].

3.1 Basic abstractions

In reactive programming languages and frameworks two distinguishing features or basic abstractions are used, behaviors and events. As defined in [Bainomugisha et al. 2013]:

Behaviors Behaviors change over time and can depend on each other. Behaviors are referred to *time-varying values* and also as *signals*. The variables in the previous example are behaviors.

Events In contrast to behaviors, events do not change, they occur at discrete points. They form a (potentially infinite) stream of values.

In reactive programming languages and frameworks these two abstractions are often realized as first-class values. In addition, these first-class values are composable via provided primitive combinators [Bainomugisha et al. 2013].

3.2 Functional reactive programming (FRP)

Reactive programming has its origin in FRP. FRP was first introduced by [Elliott and Hudak 1997] as Functional reactive animations for the programming language Haskell. The concept of behaviors of time-varying values is adapted from FRP. In contrast to reactive programming, FRP is strictly functional and focuses on modeling time.

3.3 Different levels of abstractions

Many implementations of rective programming slightly differ in there level of abstraction. Some are restricted to functional or imperative programming, and yet other support both[Maier et al. 2010]. A further partition of the base abstraction *event* into one ore many events, can also be found [pro 2017; rea 2018].

The Evaluation model, which describes how the changes are propagated, can also differ. Some implementations uses a *pull-based* approache, where the required computation needs to pull the value from the Source. In the *push-based* approache, on the other hand the source pushes changes as soon as new data arrives. Further informations and advantages and disadvantages about the two evaluation models can be found in [Bainomugisha et al. 2013].

4 SCALA.REACT

Scala.React [Maier and Odersky 2012] is a language extension to provide composable reactive programming abstractions. As the name already indicates, it is for the statically typed Scala programming language. Therefore, it provides a typesafe implementation of the abstractions and for the composing. Scala.React supports imperative as well as funtional styled programming. For the change propagation the push-based approach is used.

4.1 First class events and signals

4.2 Embedded dataflow language

4.3 Reactor

based on ..[Coplien and Schmidt 1995]

```
var path: Path = null
val moveObserver = { (event: MouseEvent) =>
  path.lineTo(event.position)
  draw(path)
}
control.addMouseDownObserver { event =>
  path = new Path(event.position)
  control.addMouseMoveObserver(moveObserver)
}
control.addMouseUpObserver { event =>
  control.removeMouseMoveObserver(moveObserver)
  path.close()
  draw(path)
}
```

Listing 1. Observer pattern

```
Reactor.loop { self =>
  // step 1
  val path = new Path((self await
       mouseDown).position)
  self.loopUntil(mouseUp) { // step 2
    val m = self awaitNext mouseMove
    path.lineTo(m.position)
    draw(path)
  }
  path.close() // step 3
  draw(path)
}
```

Listing 2. Scala.React

4.4 Glitch avoidance

5 EVOLUTION OF REACTIVE PROGRAMMING FRAMEWORKS

Since 2010, when [Maier et al. 2010] was published, reactive programming, frameworks for reactive programming and even languages evolved. Recent initiatives like the *Reactive Manifesto* [Bonér et al. 2014] and *Reactive Streams* [rea 2017] shows the gaining interest from the software community and practitioners. Research-driven reactive frameworks such as Flapjax [Meyerovich et al. 2009], Scala.React [Maier and Odersky 2012] and REScala [Salvaneschi et al. 2014] embeds all of the reactive abstractions into existing languages. Whereas

frameworks developed by communities and practitioners usually either events or behaviors implement.

with lambda-expressions and stream java added data-flow elements to the language for more declarative programming added in java 8 and already extended in java 9

5.1 Reactive databinding

The abstraction of behaviors or time-varying values is adapted by many GUI releated frameworks. These frameworks are often inspired by Flapjax and there reactivity features can be classified as reactive databinding [Maier et al. 2010].

```
IntegerProperty timeProperty = timeService.timeProperty();
StringBinding timeFormatted = Bindings.createStringBinding(
  () -> DATEFORMAT.format(timeProperty.get()),
  timeProperty);
  clock.textProperty().bind(timeFormatted);
```

Listing 3. JavaFX-properties: The time, provided by the `timeService` is bound to a formatter-binding and the formatted time is then bound to a UI-Label called `clock`

The listing 3 shows an JavaFX example. In JavaFX the term *property* is used for the abstraction, instead of behavior. The example also shows JavaFX's lack of first-class abstractions, which limits the possibilities of composing multiple properties. No declarative *map*-operator is available to transform the format of the time.

Other popular front-end frameworks besides JavaFX with reactive databinding functions are Metor, React.js and Bacon.js.

5.2 Reactive streams

6 CONCLUSION

reactive programming gained popularity also spread to other fields like cloud computing and big data [Salvaneschi et al. 2015]

glitch avoidance not always important. only in singlethreaded push based enviroments like gui-threads

REFERENCES

2017. Project Reactor. (2017). Retrieved April 17, 2018 from https://projectreactor.io/

2017. Reactive Streams. (2017). Retrieved April 17, 2018 from http://www.reactive-streams.org/

2018. ReactiveX. (2018). Retrieved April 17, 2018 from http://reactivex.io/

Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The Reactive Manifesto. (Sept. 2014). Retrieved April 17, 2018 from https://www.reactivemanifesto.org/

Edited Jim Coplien and Douglas C Schmidt. 1995. Reactor-an object behavioral pattern for demultiplexing and dispatching handles for synchronous events. (1995).

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. *SIGPLAN Not.* 32, 8 (Aug. 1997), 263–273. https://doi.org/10.1145/258949.258973

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ingo Maier and Martin Odersky. 2012. *Deprecating the Observer Pattern with Scala. React.* Technical Report.

Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. Deprecating the Observer Pattern. (2010), 18.

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. *SIGPLAN Not.* 44, 10 (Oct. 2009), 1–20. https://doi.org/10.1145/1639949.1640091

Oracle. 2017. Oracle docs: Class Observable. (April 2017). Retrieved April 17, 2018 from https://docs.oracle.com/javase/7/docs/api/java/util/Observable.html

Sean Parent. 2007. A possible future of software development. In *Talk at BoostCon*. https://stlab.cc/legacy/figures/Boostcon_possible_future.pdf

Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. 2015. Reactive Programming: A Walkthrough. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 953–954. http://dl.acm.org/citation.cfm?id=2819009.2819226

G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering* 43, 12 (Dec 2017), 1125–1143. https://doi.org/10.1109/TSE.2017.2655524

Received April 2018