Imperative Reactive Programming

The Observer Pattern

- We have seen that event handling played an important role in simulation and now we are going to see it in user interfaces
- The traditional way to deal with user interfaces to handle events there is based on the observer pattern
- The observer pattern is commonly used when we have some sort of model that maintains the state of an application and we need to have one or more views that essentially present the properties of the model in some way
- Variants of the observer pattern are also called **publish/subscribe** or **model/view/controller**
- The idea is always that we have some sort of model which captures the state of the application, and we might have one or more views that present that state and there would be a bearing numbers of views
- Views can announce themselves to the model using the method subscribe. When a change occurs to the model, it announces all the subscribed views using the method publish. And because views announce themselves as published, there can be more than one subscribed to a model
- Here is the implementation of a publisher trait:

```
trait Publisher {
   private var subscribers: Set[Subscriber] = Set()
   def subscribe(subscriber: Subscriber): Unit = subscribers += subscriber
   def unsubscribe(subscriber: Subscriber): Unit = subscribers -= subscriber
   def publish(): Unit = subscribers.foreach(_.handler(this))
}
```

- Publishers will inherit the Publisher trait. They will maintain internally a set of subscribers. All publishers should have a publish method, which is goes through all subscribers and invokes for each one a handler method that the subscriber must provide with the current publisher as its argument
- Here is the implementation of a subscriber trait:

```
trait Subscriber {
   def handler(pub: Publisher)
}
```

Observing Bank Accounts

- Let’s make BankAccount a Publisher:

```
class BankAccount extends Publisher {
    private var balance = 0
    def currentBalance: Int = balance
    def deposit(amount: Int): Unit =
        if (amount > 0) balance = balance + amount; publish()
    def withdraw(amount: Int): Unit =
        if (0 < amount && amount <= balance)
            balance = balance – amount; publish()
        else throw new Error(”insufficient funds”) }
```

- The Consolidator class is a subscriber. It observes a list of bank accounts that would always be up to date with the total balance of all accounts:

```
class Consolidator(observed: List[BankAccount]) extends Subscriber {
   private var total: Int = sum()
   private def sum() = observed.map(_.currentBalance).sum
   def handler(pub: Publisher) = sum()
   def totalBalance = total
 }
```

The Good Parts of the Observer Pattern

- Decouples views from state
- Allows to have a varying number of views of a given state
- Simple to set up

The Bad Parts of the Observer Pattern

- Forces imperative style, since handlers are Unit-typed
- Many moving parts that need to be coordinated
- Concurrency makes things more complicated
- Views are still tightly bound to one state; view update happens immediately