Dry Software Design HW2

Implementing of Observable as Monad:

```
class ObservableMonad<T, R> constructor(private val value: T) {
    private val observers = mutableListOf<(T) -> R>()

    companion object {
        @JvmStatic //we use the ctor to hide the value in a
wrapper
        fun <T, R> of(value: T): ObservableMonad<T, R> =
ObservableMonad(value)
    }

    //flat and wrap in other wrapper
    fun <S, SR> flatMap(functor: (T) -> ObservableMonad<S, SR>) =
functor(value)

    fun addObserver(callback: (T) -> R): ObservableMonad<T, R> {
        observers.add(callback)
        return this
    }

    fun removeObserver(callback: (T) -> R): ObservableMonad<T, R>
{
        observers.remove(callback)
        return this
    }

    fun notify(x: T): ObservableMonad<T, R> {
        observers.forEach { observer -> observer(x) }
        return this
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is ObservableMonad<*, *>) return false

        if (value != other.value) return false

        return true
    }

    override fun hashCode(): Int {
        var result = value?.hashCode() ?: 0
        result = 31 * result + observers.hashCode()
        return result
    }

}
```

Proving the Monoid Laws:

```
import org.junit.jupiter.api.Test
internal class ObservableMonadTest {
    private fun f(x: Long): ObservableMonad<Long, Int> =
ObservableMonad.of(x * x)
    private fun g(x: Long): ObservableMonad<Long, Int> =
ObservableMonad.of(x * x * x)
    /**
     * Law 1: Left identity
     * The first monad law states that if we take a value, put it
in a default context with return and
     * then feed it to a function by using >>=, it's the same as
just taking the value and applying
     * the function to it. To put it formally:
     *
     *      return x >>= f is the same damn thing as f x
     *
     * http://learnyouahaskell.com/a-fistful-of-monads#monad-laws
     */
    @Test
    fun leftIdentity() {

        val x = 5L
        val lhs = ObservableMonad.of<Long, Int>(x).flatMap { x ->
f(x) }
        val rhs = f(x)
        assert(lhs == rhs)
    }

    /**
     * Law 2: Right Identity
     * The second law states that if we have a monadic value and
we use >>= to feed it to return,
     * the result is our original monadic value. Formally:
     *
     *     m >>= return is no different than just m
     *
     * http://learnyouahaskell.com/a-fistful-of-monads#monad-laws
     */
    @Test
    fun rightIdentity() {
        val monadValue = ObservableMonad.of<Long, Int>(5)
        val rhs = monadValue.flatMap { value ->
ObservableMonad.of<Long, Long>(value) }
        assert(monadValue == rhs)
    }

    /**
     * Law 3: Associativity
     *
     * The final monad law says that when we have a chain of
monadic function applications with >>=,
     * it shouldn't matter how they're nested. Formally written:
     *     Doing (m >>= f) >>= g is just like doing m >>= (\x ->
f x >>= g)
     *
     * http://learnyouahaskell.com/a-fistful-of-monads#monad-laws
```

```kotlin
     */
    @Test
    fun associativity() {
        val monad = ObservableMonad.of<Long, Int>(5)

        val lhs = monad.flatMap { m -> f(m) }.flatMap { m -> g(m)
}
        val rhs = monad.flatMap { f(5).flatMap { x -> g(x) } }
        assert(lhs == rhs)
    }

}
```