`@ObservableObject`, `@Published`, and `objectWillChange.send()` are three property wrappers and modifiers that can be used to implement the ObservableObject pattern in SwiftUI.

@ObservableObject

The `@ObservableObject` property wrapper is used to create a class that can be observed by other objects. When the state of an ObservableObject class changes, it notifies all of its observers.

@Published

The `@Published` property wrapper is used to mark a property of an ObservableObject class as observable. This means that when the value of the property changes, all of the ObservableObject class's observers will be notified.

objectWillChange.send()

The `objectWillChange.send()` method is used to notify all of the ObservableObject class's observers that the state of the class is about to change. This can be useful for performing animations or other tasks that need to be synchronized with the state of the ObservableObject class.

Here is an example of how to use `@ObservableObject`, `@Published`, and `objectWillChange.send()` to create a simple counter view:

Swift
```swift
class CounterViewModel: ObservableObject {
    @Published var count = 0

    func increment() {
        objectWillChange.send()
        count += 1
    }

    func decrement() {
        objectWillChange.send()
        count -= 1
    }
}

struct CounterView: View {
    @ObservedObject var viewModel = CounterViewModel()
    var body: some View {
        VStack {
            Text("\(viewModel.count)")

            Button("Increment") {
                viewModel.increment()
            }

            Button("Decrement") {
                viewModel.decrement()
            }
        }
    }
}
```

In this example, the `CounterViewModel` class is an ObservableObject class. The `count` property is a Published property, which means that it is observable.

The `CounterViewModel` class also has two methods, `increment()` and `decrement()`, which increment and decrement the `count` property, respectively. Both of these methods call the `objectWillChange.send()` method before changing the `count` property. This ensures that all of the ObservableObject class's observers are notified before the state of the class changes.

The `CounterView` struct binds to the `CounterViewModel` class using the `@ObservedObject` property wrapper. This means that the `CounterView` struct will be automatically updated whenever the state of the `CounterViewModel` class changes.

When the user taps the "Increment" or "Decrement" button, the `increment()` or `decrement()` method is called, respectively. This updates the state of the `CounterViewModel` class and the `CounterView` struct is automatically updated to reflect the new state.

This is just a simple example of how to use `@ObservableObject`, `@Published`, and `objectWillChange.send()`. You can use these concepts to create more complex and powerful user interfaces.