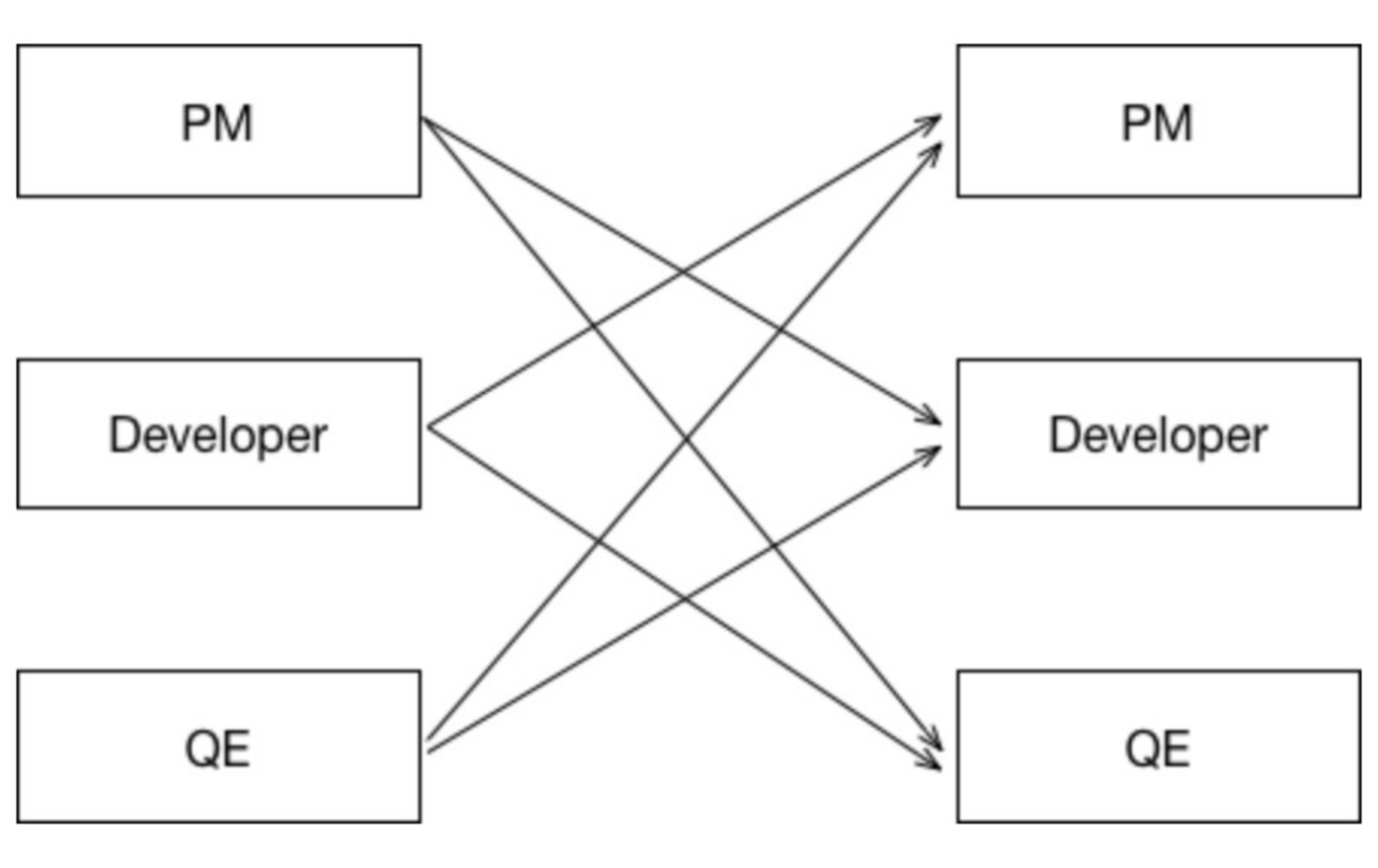
Swift World: Design Patterns — Mediator

Peng Follow
Mar 21, 2017 · 3 min read

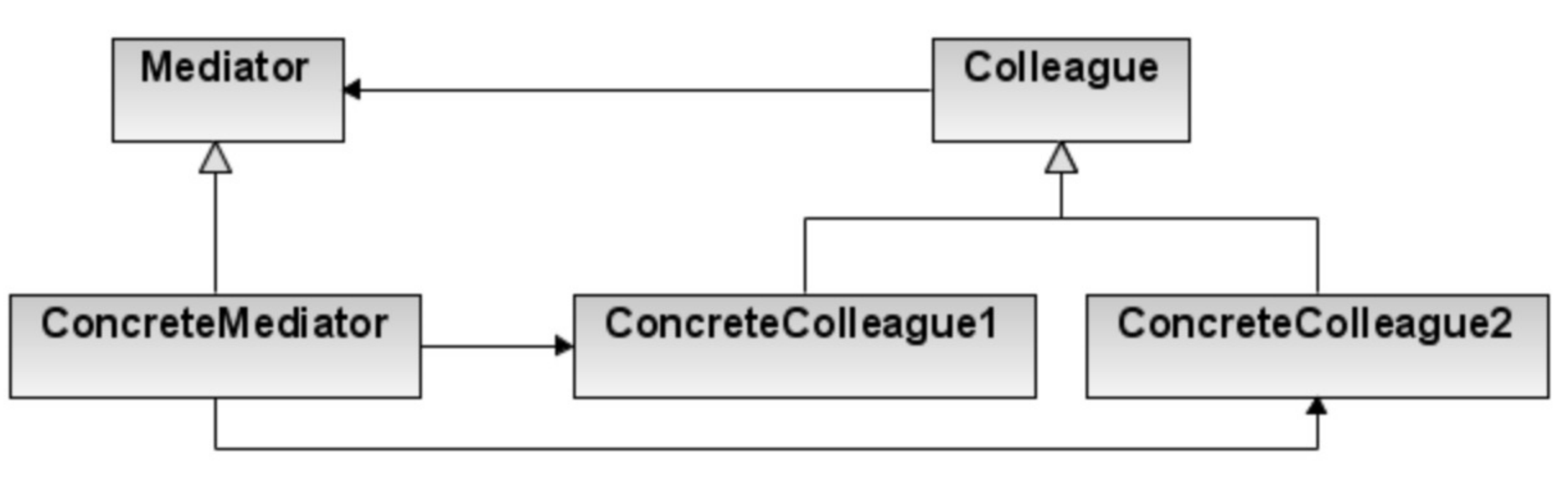
Today we will talk about mediator pattern. Let's start with a scenario in real world instead of explaining abstract definition. In a team, there are PM, developer and QE. When the developer completes coding for a new feature, the codes are committed to the repository. Other shareholders like QE and PM need to be notified.

```
protocol Collogue {
    var id: String { get }
    func send(message: String)
    func receive(message: String)
}

class Developer: Collogue {
    var id: String
    var qe: QE
    var pm: PM

    init(qe: QE, pm: PM) {
        self.id = "Developer"
        self.qe = qe
        self.pm = pm
    }

    func send(message: String) {
        qe.receive(message: message)
        pm.receive(message: message)
    }

    func receive(message: String) {
        print(message)
    }
}

class QE: Collogue {
    var id: String
    var developer: Developer
    var pm: PM

    init(developer: Developer, pm: PM) {
        self.id = "QE"
        self.developer = developer
        self.pm = pm
    }

    func send(message: String) {
        developer.receive(message: message)
        pm.receive(message: message)
    }

    func receive(message: String) {
        print(message)
    }
}

class PM: Collogue {
    var id: String
    var developer: Developer
    var qe: QE

    init(developer: Developer, qe: QE) {
        self.id = "PM"
        self.developer = developer
        self.qe = qe
    }

    func send(message: String) {
        developer.receive(message: message)
        qe.receive(message: message)
    }

    func receive(message: String) {
        print(message)
    }
}
```

Every role needs to hold instances of other roles. The connection is so tight that any changes are not easy to make.



Now, we need mediator to help us simplify the system. The mediator is defined to help objects involved to communicate with each other. It means every object interacts with mediator instead of with all others. Then the object don't need to hold reference to others but the mediator. This will decouple the system. Its structure is depicted by following figures.



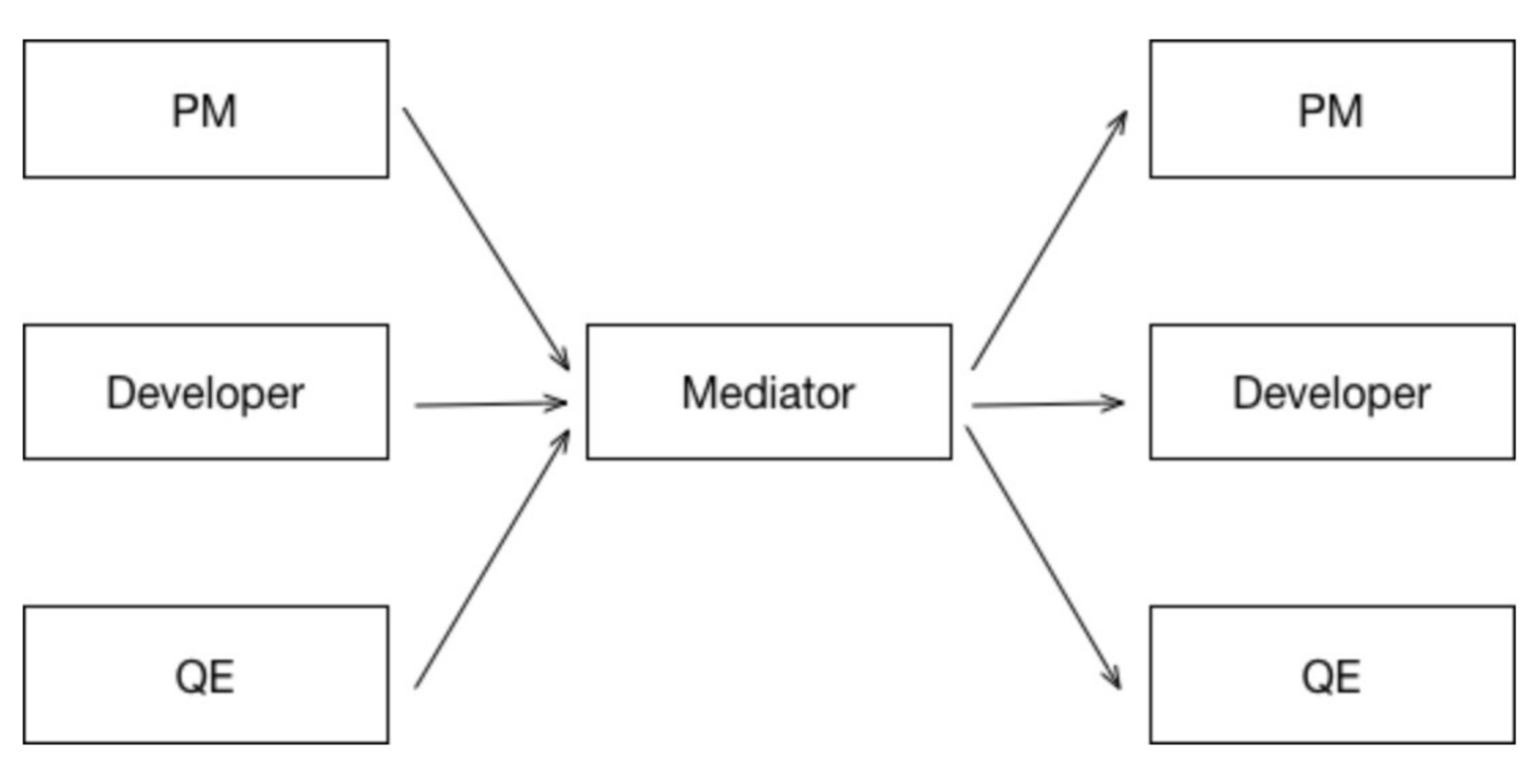
Let's start writing the code.

```
protocol Mediator {
    func send(message: String, sender: Colleague)
}

class TeamMediator: Mediator {
    var colleagues: [Colleague] = []

    func register(colleague: Colleague) {
        colleagues.append(colleague)
    }

    func send(message: String, sender: Colleague) {
        for colleague in colleagues {
            if colleague.id != sender.id {
                colleague.receive(message: message)
            }
        }
    }
}
```

With mediator, the colleagues become as following.

```
protocol Colleague {
    var id: String { get }
    var mediator: Mediator { get }
    func send(message: String)
    func receive(message: String)
}

class Developer: Colleague {
    var id: String
    var mediator: Mediator

    init(mediator: Mediator) {
        self.id = "Developer"
        self.mediator = mediator
    }

    func send(message: String) {
        mediator.send(message: message, sender: self)
    }

    func receive(message: String) {
        print("Developer received: " + message)
    }

}

class QE: Colleague {
    var id: String
    var mediator: Mediator

    init(mediator: Mediator) {
        self.id = "QE"
        self.mediator = mediator
    }

    func send(message: String) {
        mediator.send(message: message, sender: self)
    }

    func receive(message: String) {
        print("QE received: " + message)
    }
}

class PM: Colleague {
    var id: String
    var mediator: Mediator

    init(mediator: Mediator) {
        self.id = "PM"
        self.mediator = mediator
    }

    func send(message: String) {
        mediator.send(message: message, sender: self)
    }

    func receive(message: String) {
        print("PM received: " + message)
    }

}
```

So the structure becomes to the following.



Let's try it.

```
//usage
let mediator = TeamMediator()
let qe = QE(mediator: mediator)
let developer = Developer(mediator: mediator)
let pm = PM(mediator: mediator)

mediator.register(colleague: developer)
mediator.register(colleague: qe)
mediator.register(colleague: pm)

mediator.send(message: "Hello world!", sender: developer)
```

Another example is the popular NotificationCenter (aka NSNotification Center). You can find many relevant codes on the Internet.

· · ·

Thanks for your time. Please clap to get this article seen by more people. Please follow me by clicking **Follow. As** *a passionate iOS developer, blogger and open source contributor, I'm also active on* Twitter and GitHub.