Proxy Pattern

데렌 바트(Deren Bart)는 칵테일 만드는 시스템을 관리하고 있다. 이 시스템은 불편했는데, 칵테일을 만든 후 바트는 직접 손으로 쉐이커에서 남은 재료를 빼내야 했기 때문이다. 이것을 자동화하라.

페드로: 바트의 코드 베이스에 접근할 수 있나요?

이브: 아니요, 하지만 바트가 API를 몇 개 보내줬어요.

```
interface IBar {
    void makeDrink(Drink drink);
}

interface Drink {
    List<Ingredient> getIngredients();
}

interface Ingredient {
    String getName();
    double getAmount();
}
```

페드로: 바트는 소스에 손 대는 것을 원치 않아요, 대신에 IBar 인터페이스를 구현한 클래스를 추가로 몇 개 만들어서 남은 재료를 자동으로 제거할 필요가 있어요.

이브: 그러면 우리가 맡아야 할 일은 무엇인가요?

페드로: 프록시 패턴을 구현하죠. 얼마 전에 그것에 대해 읽었어요.

이브: 어서 말해 봐요.

페드로: 기본적으로 기존 모든 기능은 IBar를 구현한 표준(standard) 클래스에 위임하고, 새 기능은 ProxiedBar에 넣는 것이죠.

```
class ProxiedBar implements IBar {
    BarDatabase bar;
    IBar standardBar;

    public void makeDrink(Drink drink) {
       standardBar.makeDrink(drink);
       for (Ingredient i : drink.getIngredients()) {
           bar.subtract(i);
       }
    }
}
```

페드로: StandardBar 구현 클래스를 우리의 ProxiedBar로 바꿔야 해요.

이브: 아주 간단해 보이네요.

페드로: 네, 게다가 덤으로 우리가 기존 기능을 깨지 않아도 돼요.

이브: 확실해요? 회귀 테스트를 하지 않았어요.

페드로: 우리가 한 것은 단지 기능을 기존에 이미 테스트된 StandardBar에 위임한 것일 뿐이에요.

이브: 하지만 여전히 BarDatabase에서 쓰다 남은 재료를 제거하고 있어요.

페드로: 그것들은 분리되어 있어요.

이브: 오…

페드로: 클로저는 다른 방식이 있나요?

이브: 글쎄요, 모르겠네요. 제가 아는 바로는 함수 합성을 사용한다는 거죠.

페드로: 설명해 보세요.

이브: IBar 구현은 함수의 집합이고, 다른 IBar는 또다른 함수의 집합이죠. 당신이 추가적으로 구현해야 할 모든 것은 함수 합성으로 다 취급되요. make-drink하고, 그 다음 bar에서 subtract-ingredients 하는 것과 같은 거죠.

페드로: 코드를 보여주시는 게 더 좋지 않을까요?

이브: 네, 하지만 뭐 별로 여기서 특별한 것은 없어요.

```
;; interface
(defprotocol IBar
  (make-drink [this drink]))

;; Bart's implementation
(deftype StandardBar []
  IBar
  (make-drink [this drink]
    (println "Making drink " drink)
    :ok))

;; our implementation
(deftype ProxiedBar [db ibar]
  IBar
  (make-drink [this drink]
    (make-drink ibar drink)
    (subtract-ingredients db drink)))

;; this how it was before
(make-drink (StandardBar.)
    {:name "Manhattan"
     :ingredients [["Bourbon" 75] ["Sweet Vermouth" 25] ["Angostura" 5]]})

;; this how it becomes now
(make-drink (ProxiedBar. {:db 1} (StandardBar.))
```

```
    {:name "Manhattan"
     :ingredients [["Bourbon" 75] ["Sweet Vermouth" 25] ["Angostura" 5]]})
```

이브: 함수의 집합을 단일 객체로 그룹화하기 위해 프로토콜과 타입을 이용하죠.

페드로: 클로저가 객체지향 능력 또한 갖춘 것처럼 보이네요.

이브: 맞아요, 더욱이 reify함수가 있는데, 이것을 이용하면 runtime에 프록시를 만들 수 있어요.

페드로: 런타임용 class같은 건가요?

이브: 그 비슷하죠.

```
(reify IBar
  (make-drink [this drink]
    ;; implementation goes here
  ))
```

페드로: 간편해 보이네요.

이브: 네, 하지만 전 여전히 이것이 데코레이터 패턴과 어떻게 다른지 이해가 안돼요.

페드로: 그것들은 완전히 다르죠.

이브: 데코레이터는 같은 인터페이스에 기능을 추가하는데, 프록시도 마찬가지죠.

페드로: 글쎄요, 하지만 프록시는…

이브: 더욱이, 어댑터 또한 그리 다르지 않아요.

페드로: 어댑터는 다른 인터페이스를 이용하죠.

이브: 하지만 구현의 관점에서 보면 이런 패턴들은 모두 같아요. 즉, 어떤 것을 랩핑하고 랩퍼가 그것을 호출하죠. 그래서 “ 랩퍼 ” 가 이런 패턴들에 더 맞는 이름일 수도 있어요.

관련 문서

- Clojure Design Patterns