Javascript Design Pattern

composite Pattern

패턴의 범주별 분류

생성 관련 패턴

객체 인스턴스 생성을 위한 패턴으로, 클라이언트와 그 클라이언트에서 생성해야 할 객체 인스턴스 사이의 연결을 끊어주는 패턴

싱글턴,
추상 팩토리,
팩토리 메소드, 빌더, 프로토타입

행동 관련 패턴

클래스와 객체들의 상호작용 하는 방법 및 역할을 분담하는 방법과 관련된 패턴.

템플릿 메소드,
커맨드,
이터레이터,
옵저버,
스테이트,
스트래티지

구조 관련 패턴

클래스 및 객체들을 구성을 통해서 더 큰 구조로 만들수 있게 해주는 것과 관련된 패턴.

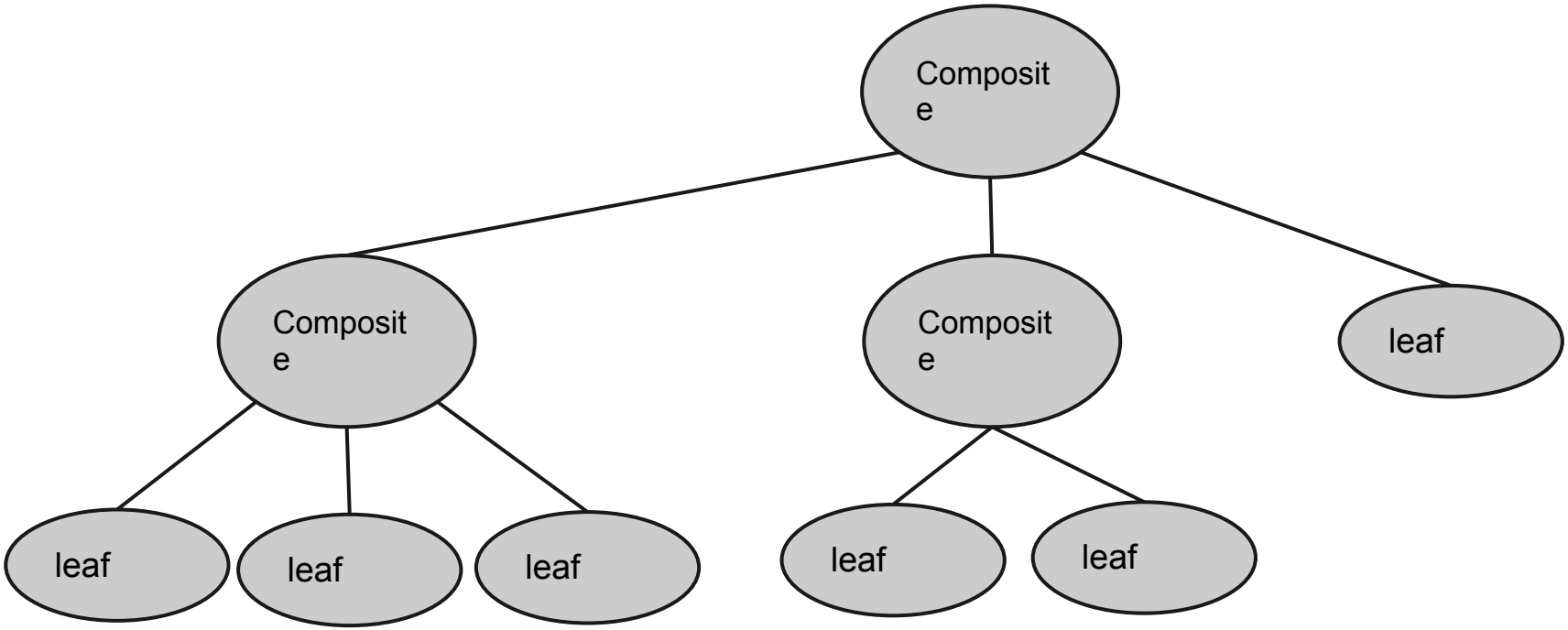
데코레이터,
컴포지트,
프록시,
퍼사드,
어댑터

Composite Pattern

- Composite Pattern을 이용하면 객체들을 트리구조로 구성하여 부분과 전체를 나타내는 계층구조로 만들 수 있습니다.

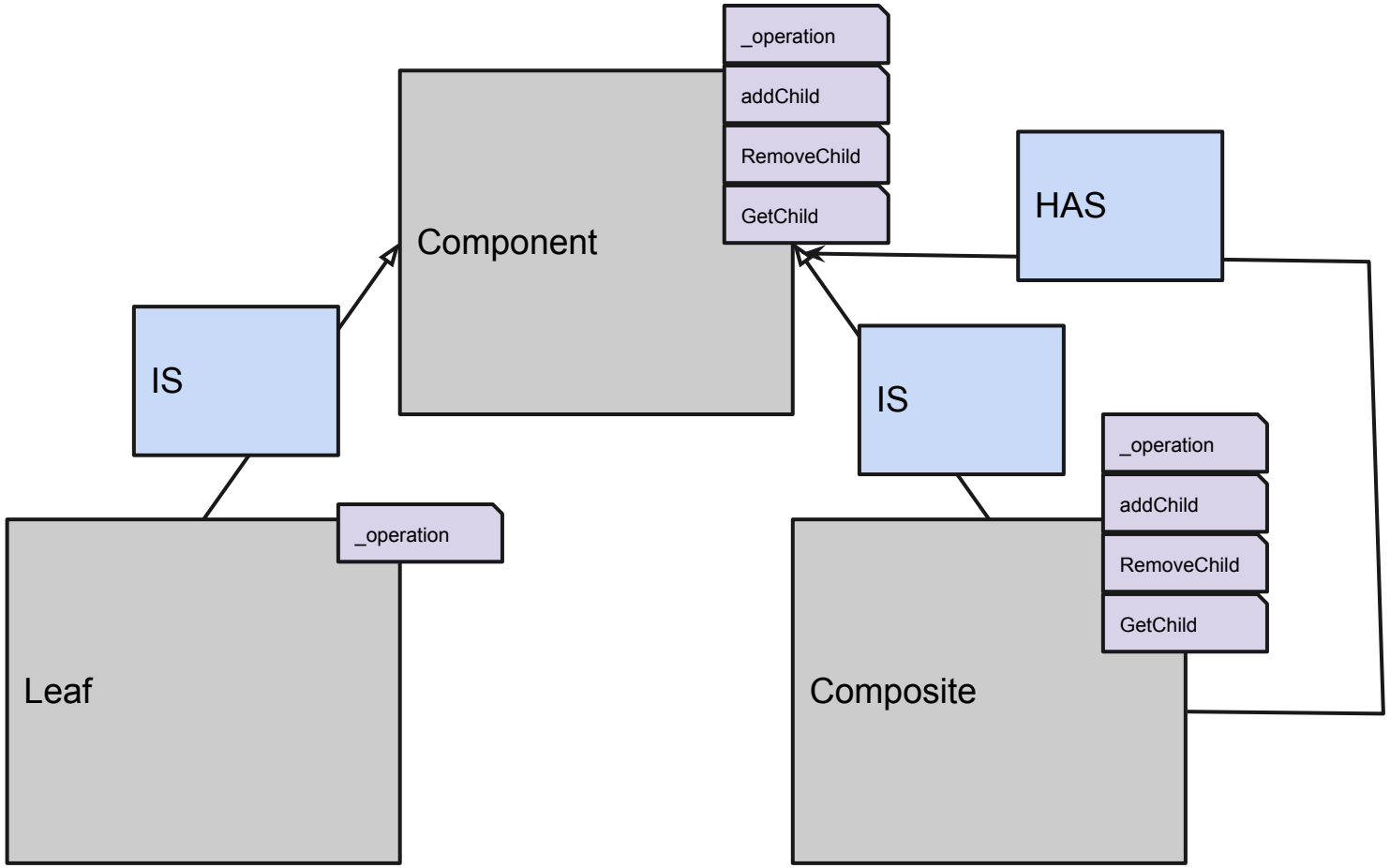
- 이 패턴을 이용하면 클라이언트에서 개별 객체와 다른 객체들로 구성된 복합객체를 똑같은 방법으로 다룰 수 있습니다.

Composite 의 구조



Composite와 Composite의 관계는 IS-A 관계가 아니라 HAS-A관계이다.

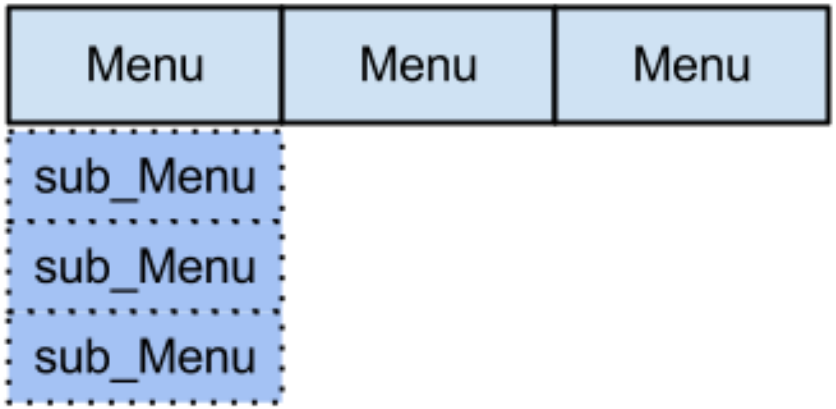
각 Composite와 Composite의 관계는 IS-A 관계가 아니라 HAS-A관계이다.



Composite Pattern Example

- Menu의 구성

- Menu 는 menuItem으로 Submenu를 가지고 Submenu또한 menuItem으로 구성되어있다.



Composite Pattern Example

- Validation Check

- Control Group은 복합 Control로 구성되어 있고, 각각의 Control또한 재귀적인 구성을 가질 수 있다.

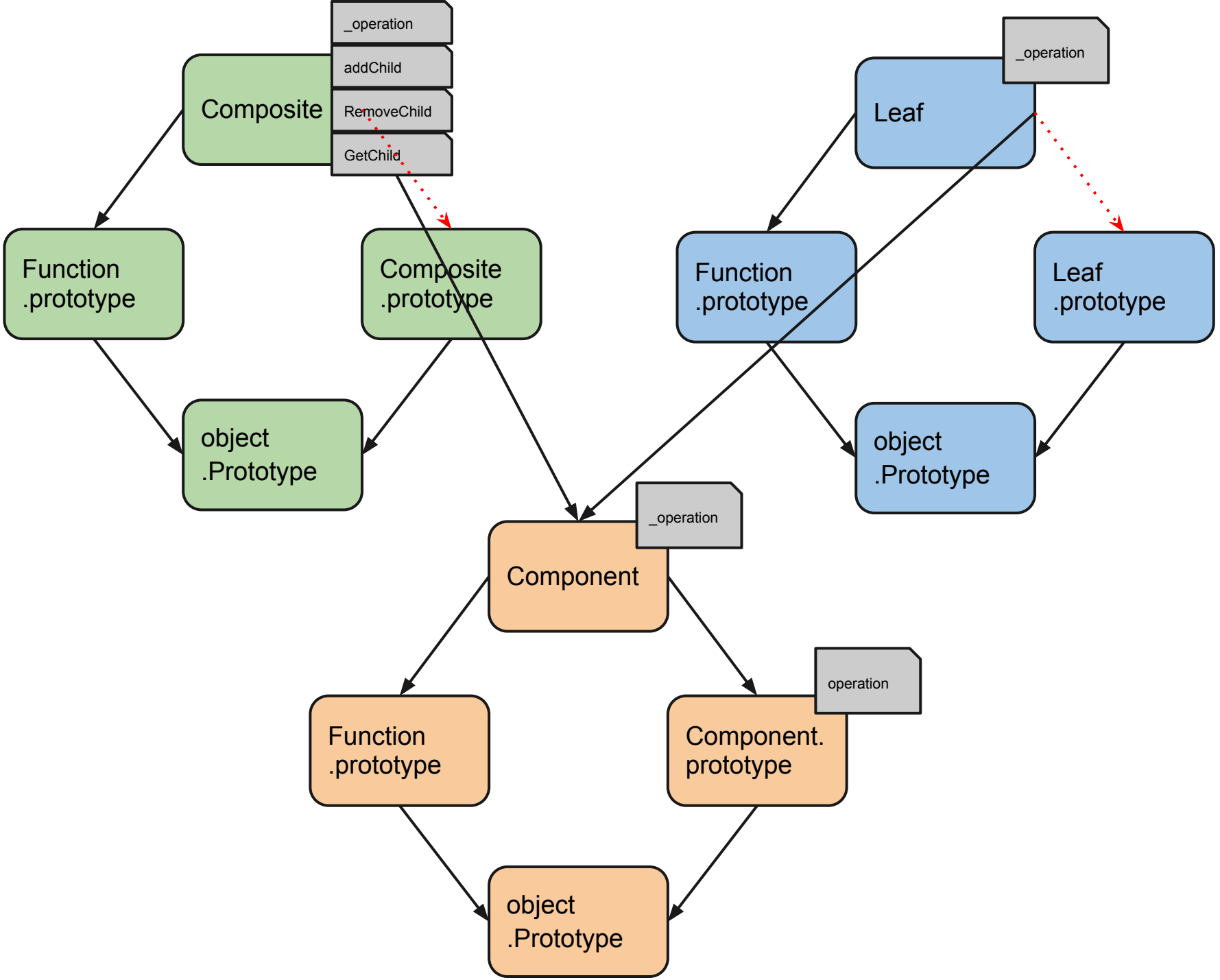
이러한 경우 Composite Pattern이 유용

```
var exemplar = function() {

    var Component = function() {
        this._Operation = function() {};
    }
    Component.prototype.Operation = function() {
        this._Operation();
    }

    var Leaf = function() {
        this._Operation = function() {
            output.writeLine("Leaf.");
        }

    }
    Leaf.prototype = new Component();
```

```
var Composite = function() {
        var _children = [];
        var self = this;
        this.AddChild = function(child) {
            _children.push(child);
        }
        this.RemoveChild = function(child) {
            throw new Error("not implemented");
        }
        this.GetChild = function(index) {
            return _children[index];
        }
        this._Operation = function() {
            output.increaseIndent();
            output.writeLine("Composite with " + _children.length + " child(ren).");
            var en = self.GetEnumerator();
            while (en.moveNext()) {
                en.current.Operation();
            }
            output.decreaseIndent();
        }
        this.GetEnumerator = function() {
            return new Enumerator(_children);
        }
    }
Composite.prototype = new Component();
```

```
var c = new Composite();
    c.AddChild(new Leaf());
    c.AddChild(new Leaf());
    c.AddChild(new Leaf());
    var b = new Composite();
    b.AddChild(new Leaf());
    b.AddChild(c);
    b.AddChild(new Leaf());
    var a = new Composite();
    a.AddChild(b);
    a.AddChild(new Leaf());
    a.Operation();
}
```

실행 FIDDLE : http://jsfiddle.net/vasco989k/BY9Hk/1/


_operation
addChild
RemoveChild
GetChild
Composite
Function
.prototype
Composite
.prototype
object
.Prototype
_operation
Leaf
Function
.prototype
Leaf
.prototype
object
.Prototype
_operation
Component
Function
.prototype
operation
Component.
prototype
object
.Prototype

Composite Pattern

- 장점

1. 반복적으로 작업을 호출할 때 특히 유용
2. 복합 내의 객체는 매우 느슨하게 결합
3. 코드 재사용을 향상시키고보다 쉽게 리팩토링이 가능

- 단점

1. 복잡성이 커지면 성능 개선점을 찾기 어렵다