O'REILLY®

패턴 모음 특별판

자바를 몰라도 시작할 수 있는 **디자인 패턴 입문서**

16여 년 만의 개정을 기념해 재탄생한 **한국 특별판**

14가지 GoF 필살 패턴!

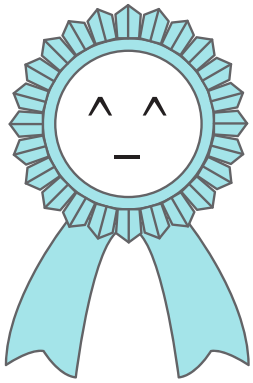
유지 관리가 편리한
객체지향 소프트웨어를 만드는 법

헤드퍼스트 디자인패턴

개정판

에릭 프리먼·엘리자베스 롭슨·케이시 시에라·버트 베이츠 지음 | 서환수 옮김

한빛미디어 Hanbit Media, Inc.

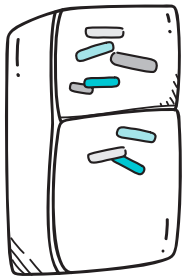


헤드 퍼스트 디자인 패턴
개정판에 관심 주셔서
감사합니다!

14가지 핵심 GoF 패턴과 비교해서 조금 덜 쓰일 뿐
상황에 따라 유용하게 사용할 수 있는 9가지 패턴을 준비했습니다.

앞으로도 많은 관심 부탁드립니다!

GoF 패턴 중에
나쁜 패턴은 없습니다!



여러분의 두뇌를 정복하는 방법

이제 여러분이 행동할 차례입니다. 여기에 나와 있는 팁에서부터 시작합시다. 여러분의 두뇌에서 어떤 반응을 보이는지 살펴보고 어떤 것이 적절하고 어떤 것이 부적절한지 알아봅시다. 항상 새로운 것을 시도해 보세요.

아래 내용을 오려서 냉장고 문에 붙여 놓으세요.

---

1 천천히 하세요. 더 많이 이해할수록 외워야 할 양은 줄어듭니다.

그냥 무작정 읽지 맙시다. 잠깐씩 쉬면서 생각해 봅시다. 책에 있는 질문을 보고 정답으로 바로 넘어가면 안 됩니다. 누군가 다른 사람이 정말로 질문을 하고 있다고 생각하세요. 더 깊이, 신중하게 생각할수록 더 잘 배우고 기억할 수 있습니다.

2 연습문제를 꼭 풀어 보고 메모를 남깁시다.

연습문제는 여러분을 위해 수록한 것입니다. 그냥 답만 보고 넘어가면 다른 사람이 운동하는 걸 구경하는 것과 마찬가지입니다. 연습문제를 눈으로만 보고 넘어가면 안 됩니다. 반드시 직접 필기구를 들고 문제를 풀어 봅시다. 실제로 배우는 과정에서 몸을 움직이는 것이 배우는 데 도움이 된다고 합니다.

3 〈무엇이든 물어보세요〉 부분을 반드시 읽어 봅시다.

반드시 모두 읽어 보세요. 그냥 참고자료로 수록한 것이 아니라 이 책의 핵심 내용 중 하나입니다.

4 잠자리에 들기 전에 마지막으로 이 책을 읽어 봅시다.

학습 과정의 일부(특히 장기 기억으로의 전이 과정)는 책을 놓은 후에 일어납니다. 여러분의 두뇌에서 무언가를 처리하려면 시간이 필요하기 때문이죠. 처리하는 중에 다른 일을 하면 새로 배운 내용을 잊어버릴 가능성이 높아집니다.

5 물을 많이 마십시다.

머리가 잘 돌아가려면 물이 많이 필요합니다. 수분이 부족하면 (목이 마르다는 느낌이 들면 수분이 부족한 것입니다) 인지 기능이 떨어집니다.

6 배운 내용을 얘기해 봅시다.

소리 내어 말하면 읽기만 할 때와는 다른 두뇌 부분이 활성화됩니다. 뭔가를 이해하거나 나중에 더 잘 기억하고 싶다면 크게 소리를 내어 말해 보세요. 다른 사람에게 설명하면 더 좋습니다. 더 빠르게 배울 수 있으며 책을 읽는 동안에는 몰랐던 것도 생각할 수 있습니다.

7 두뇌의 반응에 귀를 기울여 봅시다.

여러분의 두뇌가 너무 힘들어하고 있지는 않은지 관심을 가져봅시다. 대강 훑어보고 있거나 방금 읽은 내용을 바로 잊어버린다는 느낌이 들면 잠시 쉬는 것이 좋습니다. 일단 어느 정도 공부하고 나면 무조건 파고든다고 해서 더 빨리 배울 수 있는 것은 아닙니다. 오히려 공부하는 데 방해가 될 수도 있습니다.

8 뭔가를 느껴 봅시다.

여러분의 두뇌에서 지금 공부하고 있는 것이 중요하다고 느낄 수 있어야 합니다. 책 속에 나와 있는 이야기에 몰입해 보고 책에 나와 있는 사진에 직접 제목을 붙여 봅시다. 아무것도 느끼지 않는 것보다는 썰렁한 농담을 보고 비웃기라도 하는 편이 낫습니다.

9 직접 디자인해 봅시다.

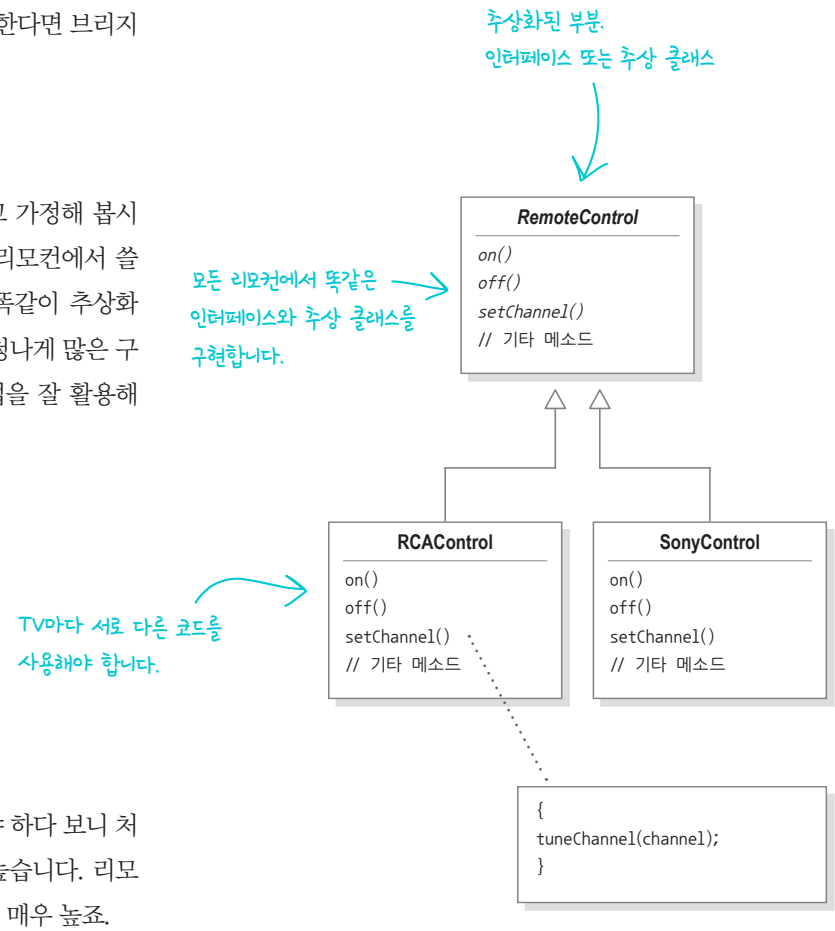
여러분의 디자인에 새로운 것을 적용해 보거나 기존 프로젝트를 리팩터링해 봅시다. 이 책에 나와 있는 연습문제 같은 것 외에 뭔가 경험이 될 수 있는 것이라면 뭐든지 좋습니다. 여러분에게 필요한 것은 연필과 해결해야 할 문제뿐입니다. 물론 디자인 패턴을 적용해서 해결하기 좋은 문제여야겠죠.

브리지 패턴

구현과 더불어 추상화 부분까지 변경해야 한다면 브리지(Bridge) 패턴을 쓰면 됩니다.

시나리오

혁신적인 '만능 리모컨'을 만들기로 했다고 가정해 봅시다. 인체공학적이고 사용자 친화적인 TV 리모컨에서 쓸 코드를 만들어야 합니다. 리모컨 자체는 똑같이 추상화 부분을 바탕으로 하지만, TV 모델마다 엄청나게 많은 구현 코드를 사용해야 하므로 객체지향 기법을 잘 활용해야 합니다.

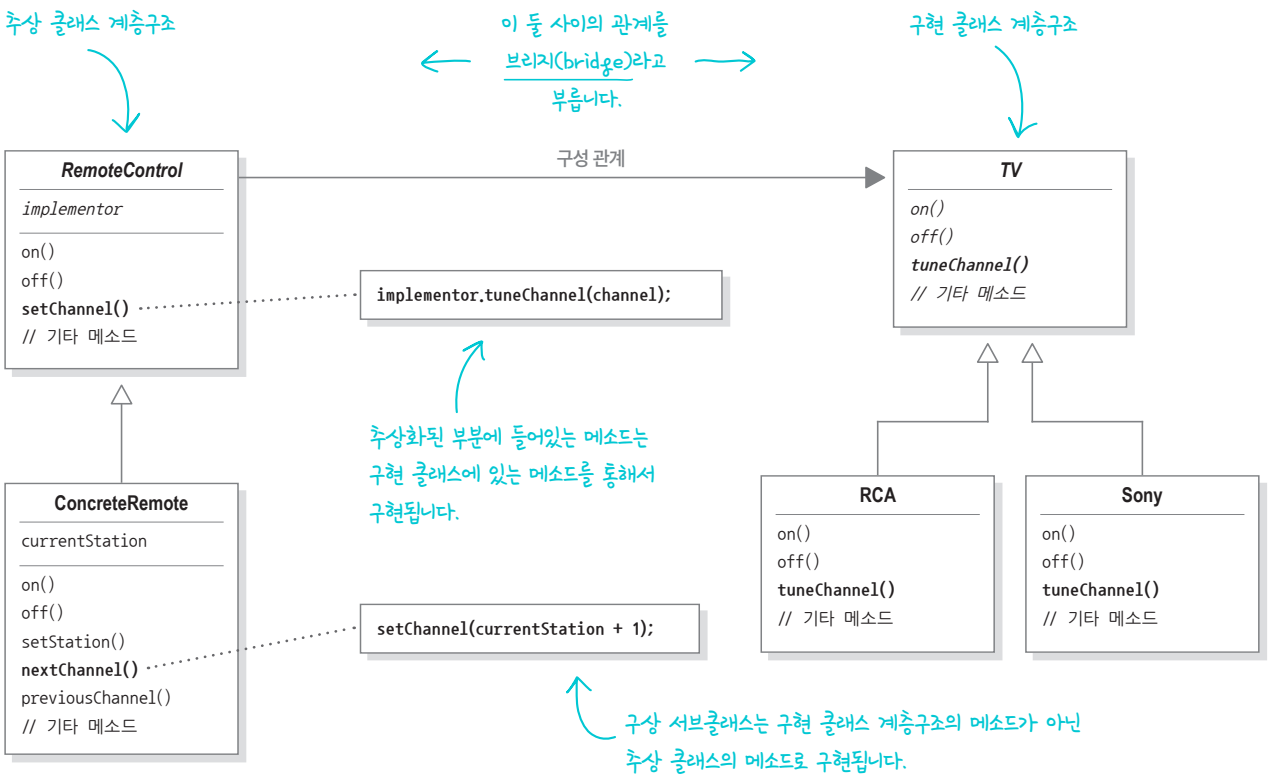


해야 할 일: 추상화 부분도 바꿔야 합니다.

이 리모컨은 모든 TV를 대상으로 작동해야 하다 보니 처음에는 제대로 작동하지 않을 가능성이 높습니다. 리모컨의 기능을 여러 번 다듬어야 할 가능성이 매우 높죠.
따라서 리모컨도 바뀔 수 있고, TV도 바뀔 수 있다는 딜레마에 직면하게 됩니다. 사용자 인터페이스는 이미 추상화했으므로 리모컨 사용자가 쓸 다양한 TV 종류에 따라 구상 클래스를 바꿔 쓸 수 있습니다. 하지만 사용자들이 제공하는 정보에 맞춰서 리모컨을 개선하다 보면 추상화 부분까지도 바꿔야 할 수도 있습니다.
이렇게 구체적인 구현 부분과 추상화 부분을 모두 바꿀 수 있는 객체지향 디자인을 어떻게 만들어야 할까요?

브리지 패턴 사용하기

브리지 패턴을 사용하면 추상화된 부분과 구현 부분을 서로 다른 클래스 계층구조로 분리해서 그 둘을 모두 변경할 수 있습니다.



이렇게 2개의 계층구조를 만들었습니다. 한쪽은 리모컨을 나타내는 부분이고 다른 쪽은 종류별로 다른 TV를 나타내는 부분이죠. 하지만 브리지로 연결했으므로 양쪽을 서로 독립적으로 바꿔 줄 수 있습니다.

브리지 패턴의 장점

- 구현과 인터페이스를 완전히 결합하지 않았기에 구현과 추상화 부분을 분리할 수 있습니다.
- 추상화된 부분과 실제 구현 부분을 독립적으로 확장할 수 있습니다.
- 추상화 부분을 구현한 구상 클래스가 바뀌어도 클라이언트에는 영향을 끼치지 않습니다.

브리지 패턴의 활용법과 단점

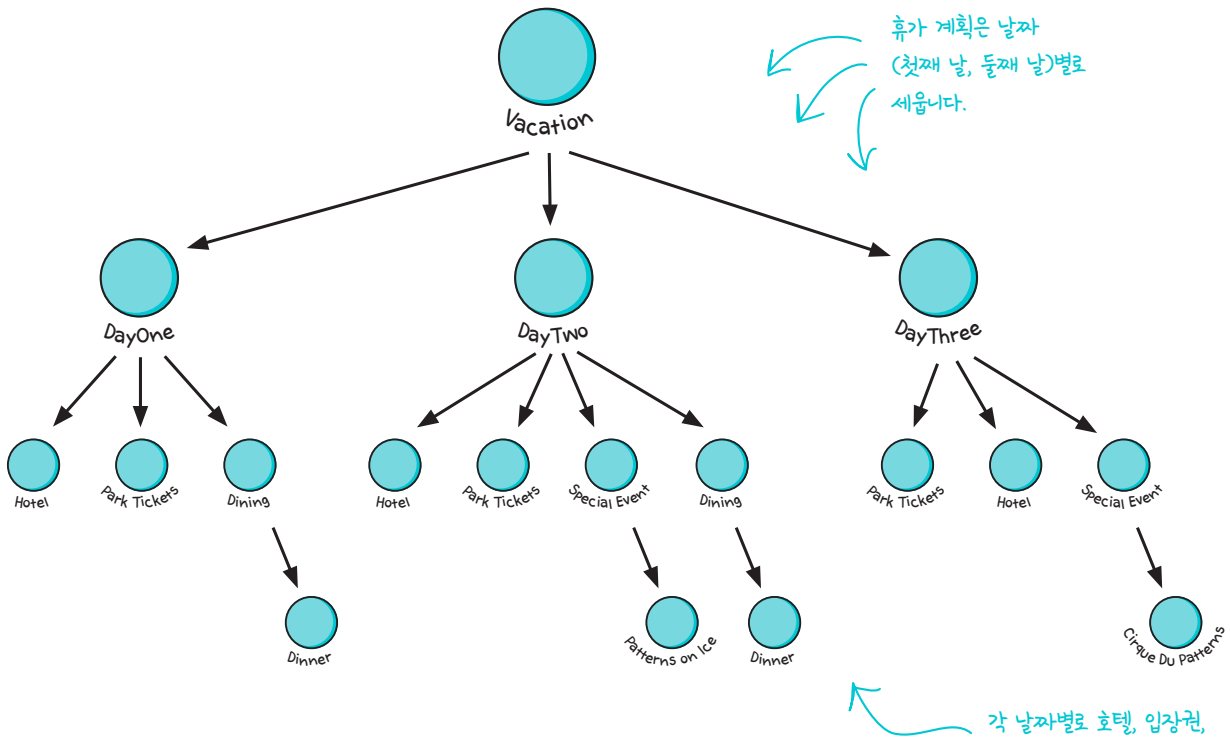
- 여러 플랫폼에서 사용해야 하는 그래픽스와 윈도우 처리 시스템에서 유용하게 쓰입니다.
- 인터페이스와 실제 구현할 부분을 서로 다른 방식으로 변경해야 할 때 유용하게 쓰입니다.
- 디자인이 복잡해진다는 단점이 있습니다.

빌더 패턴

제품을 여러 단계로 나눠서 만들도록 제품 생산 단계를 캡슐화하고 싶다면 빌더(Builder) 패턴을 사용하면 됩니다.

시나리오

객체마을 외곽에 새로 생길 패턴랜드 테마 파크에서 고객에게 제공할 휴가 계획 프로그램을 만들어야 합니다. 패턴랜드 고객은 호텔, 입장권, 레스토랑, 특별 이벤트 등을 마음대로 선택해서 예약할 수 있습니다. 휴가 계획 프로그램을 만들려면 다음과 같은 구조가 필요합니다.



해야 할 일: 유연한 디자인이 필요합니다

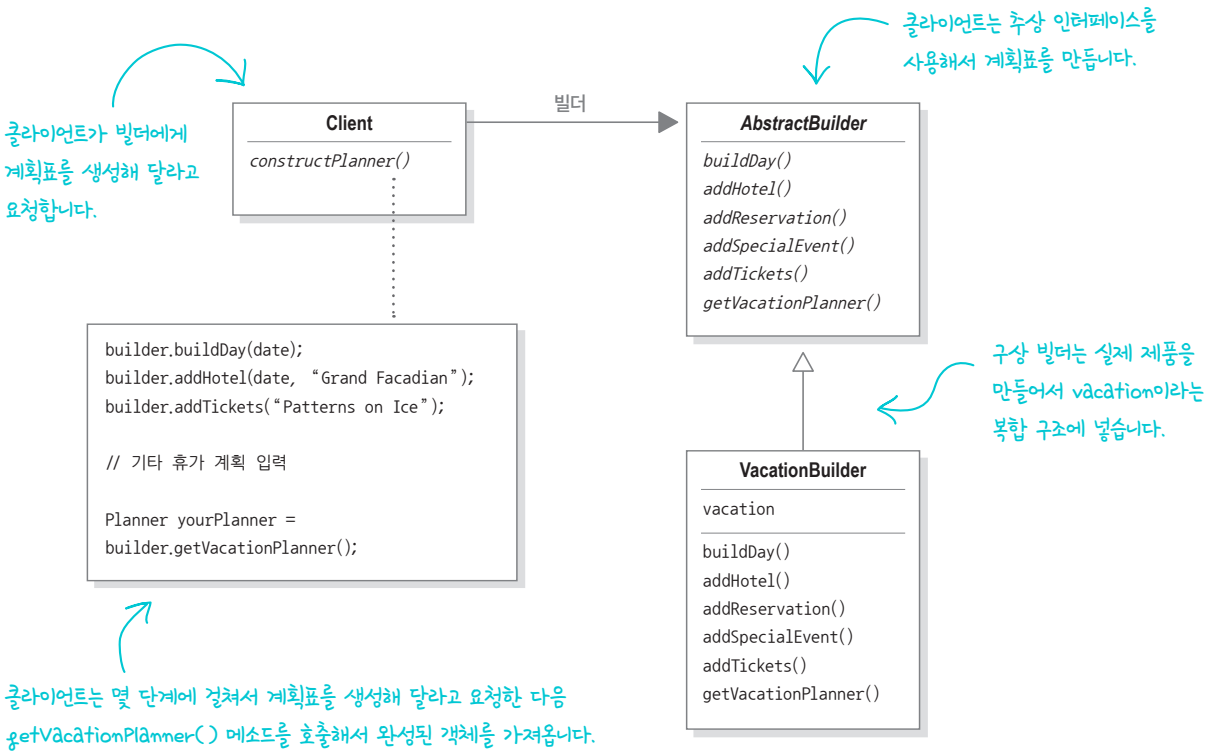
손님마다 휴가 일수와 하는 일이 다릅니다. 예를 들어 객체마을 주민이라면 호텔은 예약하지 않는 대신, 저녁 식사와 특별 이벤트만 예약할 수도 있습니다. 반대로 멀리서 놀러 온 관광객들은 호텔, 저녁 식사, 입장권을 모두 예약하겠죠.

따라서 다양한 손님의 계획표를 표현할 수 있는 유연한 자료구조가 필요합니다. 계획표를 만들려면 꽤 복잡한 단계를 거쳐야 할 수도 있습니다.

어떻게 하면 복잡한 계획표와 계획표를 만드는 단계가 서로 섞이지 않게 하면서 계획을 짤 수 있을까요?

빌더 패턴 사용하기

혹시 반복자 패턴을 기억하나요? 그 패턴을 사용하면 반복 작업을 별도의 객체로 캡슐화해서 컬렉션의 내부 구조를 클라이언트로부터 보호할 수 있습니다. 여기에서도 똑같은 아이디어를 사용합니다. 계획표 작성을 객체(빌더라고 부름)에 캡슐화해서 클라이언트가 빌더에게 계획표 구조를 만들어 달라고 요청하도록 만드는 거죠.



빌더 패턴의 장점

- 복합 객체 생성 과정을 캡슐화합니다.
- 여러 단계와 다양한 절차를 거쳐 객체를 만들 수 있습니다(팩토리 패턴은 한 단계에서 모든 걸 처리하죠).
- 제품의 내부 구조를 클라이언트로부터 보호할 수 있습니다.
- 클라이언트는 추상 인터페이스만 볼 수 있기에 제품을 구현한 코드를 쉽게 바꿀 수 있습니다.

빌더 패턴의 활용법과 단점

- 복합 객체 구조를 구축하는 용도로 많이 쓰입니다.
- 팩토리를 사용할 때 보다 객체를 만들 때 클라이언트에 관해 더 많이 알아야 합니다.

책임 연쇄 패턴

1개의 요청을 2개 이상의 객체에서 처리해야 한다면 책임 연쇄(Chain of Responsibility) 패턴을 사용하면 됩니다.

시나리오

자바가 탑재된 뽑기 기계 출시 이후로, 주식회사 왕뽑기에는 감당하기 힘들 정도로 많은 이메일이 날아오기 시작했습니다. 그들의 분석에 의하면 이메일은 크게 4가지로 분류할 수 있다고 합니다. 새로 추가된 10%의 확률로 한 개 더 받을 수 있는 기능에 만족한 고객으로부터 오는 팬 메일, 아이들이 뽑기에 중독됐다면서 기계를 다른 데로 옮겨 달라고 항의하는 부모님들의 메일, 뽑기 기계를 새로 설치해 달라는 메일, 그리고 스팸 메일, 이렇게 말이죠.
팬 메일은 전부 CEO에게 직접 전달해야 하고, 항의 메일은 법무 담당 부서로, 신규 설치 요청 메일은 영업부로 전달해야 합니다. 그리고 스팸 메일은 그냥 지워야 하죠.

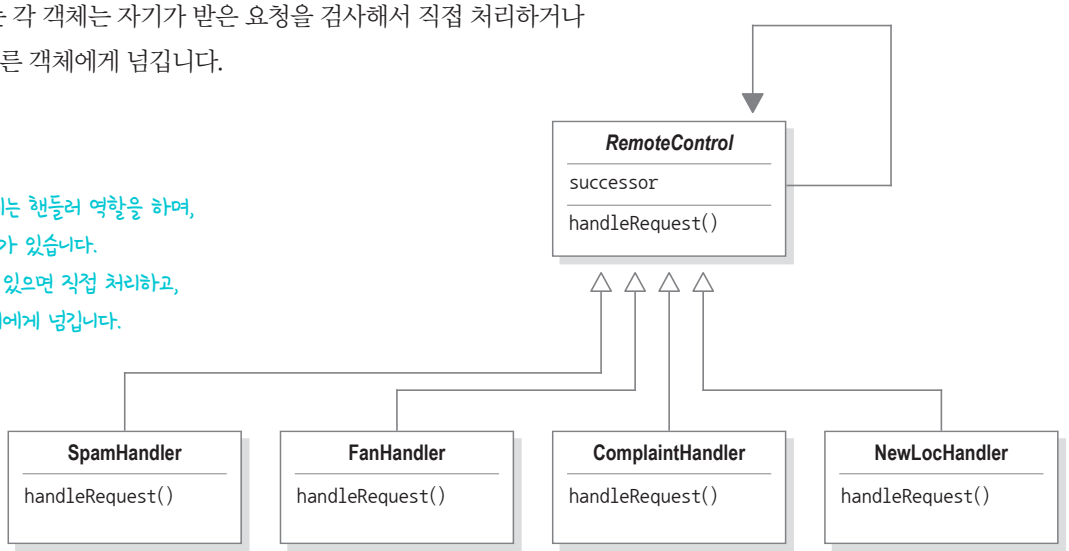
해야 할 일: 검출기로 메일을 분류해야 합니다.

주식회사 왕뽑기에서 이미 스팸 메일, 팬 메일, 항의 메일, 신규 설치 요청 메일을 감지해 주는 인공지능 검출기를 만들어 놓았습니다. 이 검출기를 써서 메일을 분류하는 디자인을 만들어야 합니다.

책임 연쇄 패턴 사용하기

책임 연쇄 패턴에서는 주어진 요청을 검토하는 객체 사슬을 생성합니다.
그 사슬에 속해 있는 각 객체는 자기가 받은 요청을 검사해서 직접 처리하거나
사슬에 들어있는 다른 객체에게 넘깁니다.

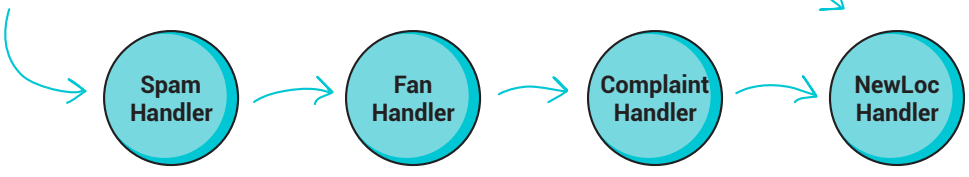
사슬에 들어있는 각 객체는 핸들러 역할을 하며,
객체마다 이어지는 객체가 있습니다.
주어진 요청을 처리할 수 있으면 직접 처리하고,
그렇지 않으면 다음 객체에게 넘깁니다.



이메일이 수신되면 첫 번째 핸들러인 SpamHandler에게 전달됩니다. SpamHandler가 처리할 수 없으면 FanHandler로 넘기죠. 이처럼 사슬을 따라 요청이 전달되면서 적절한 핸들러가 메일을 분류합니다.

모든 이메일은 우선 첫 번째
핸들러에게 전달됩니다.

사슬 맨 끝까지 간 이메일은 처리되지
못 합니다. 필요에 따라서 모든 요청을
처리하는 핸들러를 구현할 수도 있습니다.



책임 연쇄 패턴의 장점

- 요청을 보낸 쪽과 받는 쪽을 분리할 수 있습니다.
- 객체는 사슬의 구조를 몰라도 되고 그 사슬에 들어있는 다른 객체의 직접적인 레퍼런스를 가질 필요도 없으므로 객체를 단순하게 만들 수 있습니다.
- 사슬에 들어가는 객체를 바꾸거나 순서를 바꿈으로써 역할을 동적으로 추가하거나 제거할 수 있습니다.

책임 연쇄 패턴의 활용법과 단점

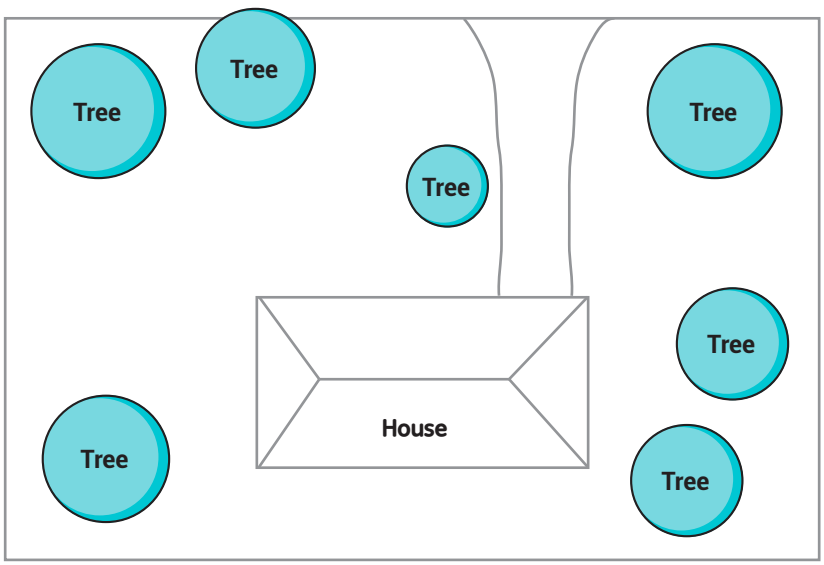
- 윈도우 시스템에서 마우스 클릭과 키보드 이벤트를 처리할 때 흔히 쓰입니다.
- 요청이 반드시 수행된다는 보장이 없다는 단점이 있습니다. 사슬 끝까지 갔는데도 처리되지 않을 수 있죠(사실 이런 특성이 장점이 될 수도 있긴 합니다).
- 실행 시에 과정을 살펴보거나 디버깅하기가 힘들다는 단점이 있습니다.

플라이웨이트 패턴

어떤 클래스의 인스턴스 하나로 여러 개의 '가상 인스턴스'를 제공하고 싶다면 플라이웨이트(Flyweight) 패턴을 사용하면 됩니다.

시나리오

조경 설계 애플리케이션에서 나무를 객체 형태로 추가해야 합니다. 애플리케이션 내에서 나무들의 역할은 별로 중요하지 않습니다. 그냥 X, Y 좌표를 가지고 있고, 나무의 나이에 따라 적당한 크기로 화면에 표현하면 됩니다. 문제는 어떤 사용자가 나무를 꽤 많이 넣으려고 한다는 점입니다. 대강 모양을 보면 다음과 같습니다.



Tree 인스턴스에는
나무의 상태가 저장됩니다.

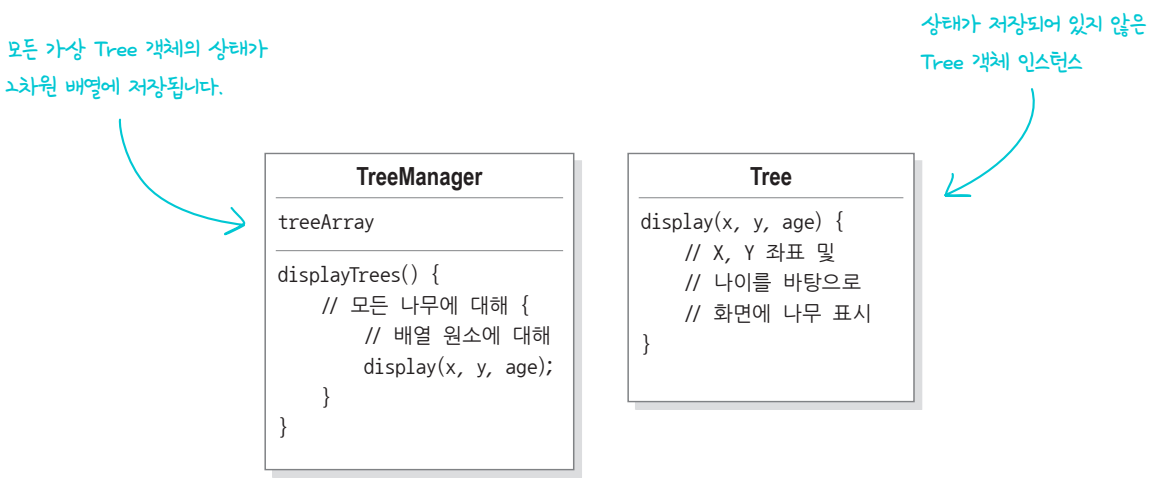
```
Tree
xCoord
yCoord
age
display() {
    // X, Y 좌표 및
    // 수령을 바탕으로
    // 화면에 나무 표시
}
```

해야 할 일: 실행 중 느려지지 않게 만들어야 합니다.

몇 달 동안 공을 들인 끝에 드디어 제법 큰 고객을 잡았습니다. 1000 카피를 구입할 예정이고, 우리가 만든 소프트웨어를 써서 대규모 주택단지의 조경을 설계하려고 합니다. 그런데 이 소프트웨어를 1주일 동안 테스트해 본 결과 나무를 많이 만들면 애플리케이션이 눈에 띄게 느려진다는 사실을 발견했다고 합니다.

플라이웨이트 패턴 사용하기

Tree 객체를 수 천 개 만드는 대신 시스템을 조금 고쳐서 Tree의 인스턴스는 하나만 만들고 모든 나무의 상태를 클라이언트 객체가 관리하도록 하면 어떨까요? 이게 바로 플라이웨이트 패턴입니다.



플라이웨이트 패턴의 장점

- 실행 시에 객체 인스턴스의 개수를 줄여서 메모리를 절약할 수 있습니다.
- 여러 '가상' 객체의 상태를 한곳에 모아 둘 수 있습니다.

플라이웨이트의 패턴 사용법과 단점

- 어떤 클래스의 인스턴스가 아주 많이 필요하지만 모두 똑같은 방식으로 제어해야 할 때 유용하게 쓰입니다.
- 일단 이 패턴을 써서 구현해 놓으면 특정 인스턴스만 다른 인스턴스와 다르게 행동하게 할 수 없다는 단점이 있습니다.

인터프리터 패턴

어떤 언어의 인터프리터를 만들 때는 인터프리터(Interpreter) 패턴을 사용하면 됩니다.

잠시 쉬어가기

인터프리터 패턴을 제대로 구현하려면 문법을 어느 정도 알고 있어야 합니다. 하지만 문법을 제대로 공부해 본 적이 없더라도 이 패턴을 이해하는 데는 무리가 없습니다. 그냥 다음 쪽에 있는 인터프리터 패턴을 계속 살펴보세요.

시나리오

오리 시뮬레이션 게임 기억하죠? 그 게임을 아이들에게 프로그래밍을 가르쳐 주는 용도로 활용하는 기막힌 아이디어가 떠올랐습니다. 아이 1명당 오리 1마리를 정해 준 다음에 간단한 언어를 가르쳐 주는 겁니다. 예를 들어 다음과 같은 식으로 말이죠.

```
right;
while (daylight) fly;
quack;
```

우회전

낮 동안 계속 날아다니게 함

꽥꽥 소리를 내게 함

옛날 옛적에 들었던 프로그래밍 기초 수업의 기억을 떠올리면서 다음과 같은 문법 규칙을 만들었습니다.

```
expression ::=  <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')'<expression>
variable ::= [A-Z,a-z]+
```

프로그램이란 일련의 명령어(command)와 시퀀스(sequence), 반복문(repetition)으로 구성된 표현식(expression)입니다.

시퀀스란 세미콜론으로 여러 표현식을 연결해 놓은 것입니다.

명령어에는 right, quack, fly, 이렇게 3가지가 있습니다.

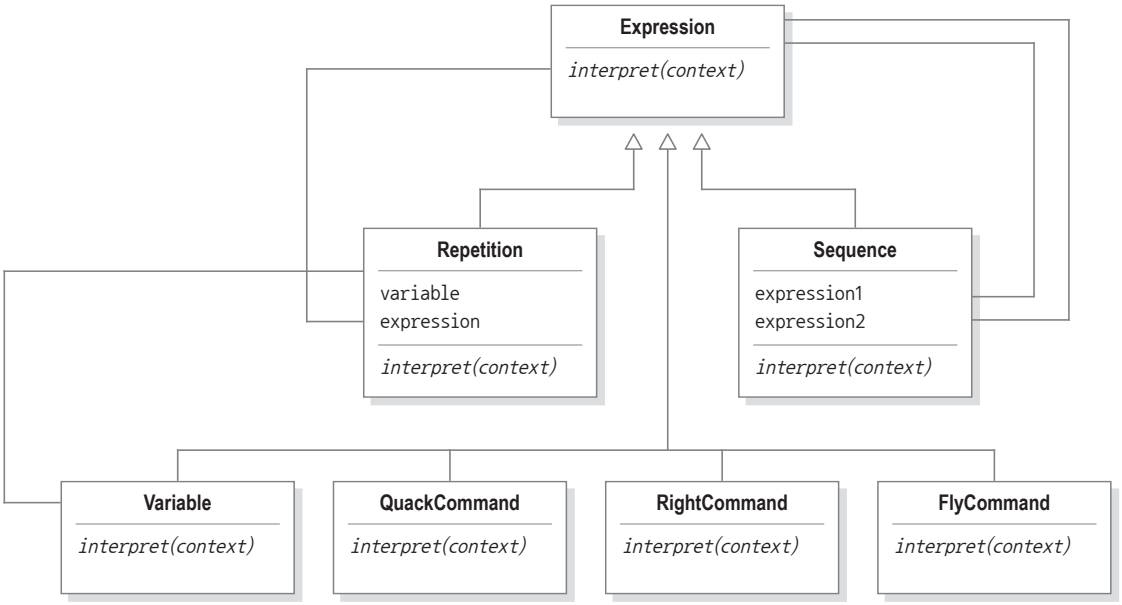
while문은 조건 변수와 표현식으로 구성됩니다.

해야 할 일: 인터프리터를 만들어야 합니다.

이제 문법을 완성했으니까 아이들이 오리의 움직임을 프로그래밍하고 그 결과를 직접 확인할 수 있도록 위의 문법에 따라 만들어진 코드 해석용 인터프리터를 만들어야 합니다.

인터프리터 패턴 사용하기

인터프리터 패턴은 문법과 구문을 번역하는 인터프리터 클래스를 기반으로 간단한 언어를 정의합니다. 언어에 속하는 규칙을 나타내는 클래스를 사용해서 언어를 표현합니다. 오리 언어를 클래스로 표현하면 다음과 같습니다. 문법과 직접적으로 대응한다는 사실을 확인할 수 있습니다.



이 언어를 해석하려면 각 표현식에 대응하는 interpret() 메소드를 호출하면 됩니다. 이 메소드에 컨텍스트(파싱하고 있는 프로그램의 입력 스트림이 들어있음)도 전달되며, 입력된 내용을 확인하고 평가하는 작업도 이 메소드가 처리합니다.

인터프리터 패턴의 장점

- 문법을 클래스로 표현해서 쉽게 언어를 구현할 수 있습니다.
- 문법이 클래스로 표현되므로 언어를 쉽게 변경하거나 확장할 수 있습니다.
- 클래스 구조에 메소드만 추가하면 프로그램을 해석하는 기본 기능 외에 예쁘게 출력하는 기능이나 더 나은 프로그램 확인 기능 같은 새로운 기능을 추가할 수 있습니다.

인터프리터 패턴의 활용법과 단점

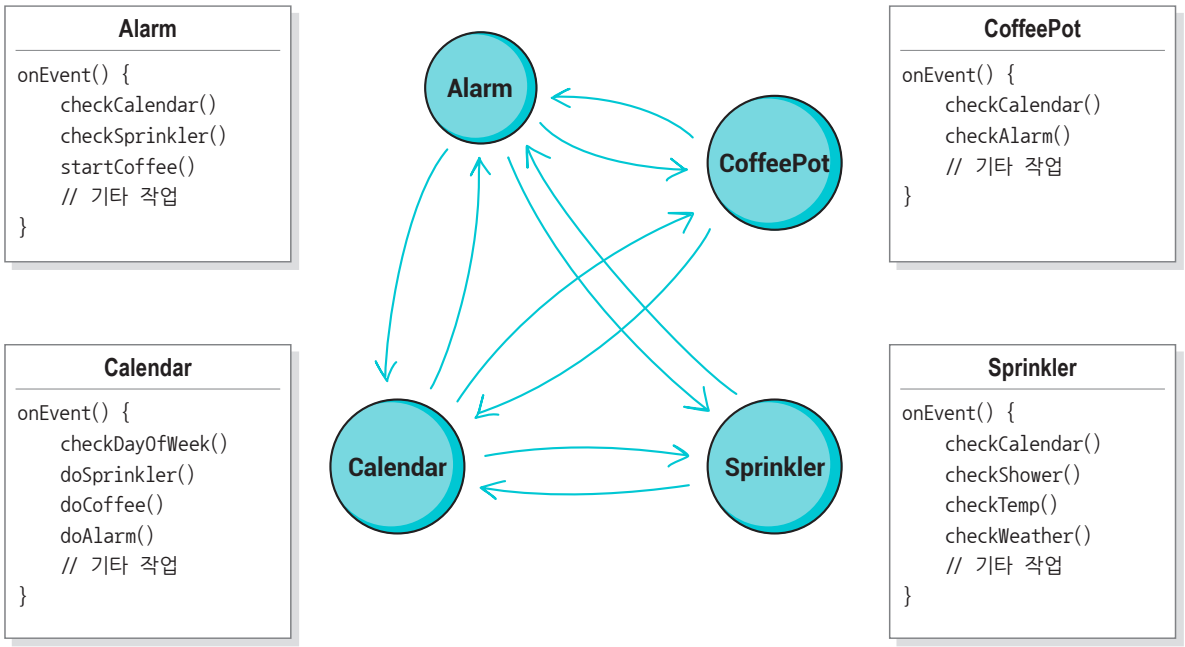
- 간단한 언어를 구현할 때 인터프리터 패턴이 유용하게 쓰입니다.
- 효율보다는 단순하고 간단하게 문법을 만드는 것이 더 중요한 경우에 유용합니다.
- 스크립트 언어와 프로그래밍 언어에서 모두 쓸 수 있습니다.
- 문법 규칙의 개수가 많아지면 아주 복잡해진다는 단점이 있습니다. 그럴 때는 파서나 컴파일러 생성기를 쓰는 편이 낫습니다.

중재자 패턴

서로 관련된 객체 사이의 복잡한 통신과 제어를 한곳으로 집중하고 싶다면 중재자(Mediator) 패턴을 쓰면 됩니다.

시나리오

상원은 미래의 집 그룹 친구들의 도움을 받아서 자바 기술을 사용하는 자동화 주택을 만들었습니다. 모든 가전제품에 자바가 내장되어 있어 정말 편해졌습니다. 상원이 더 이상 알람 시계의 5분만 더 버튼을 누르지 않으면 알람 시계에서 커피 메이커로 "커피를 만들어"라고 신호를 보내죠. 지금도 상당히 편리하긴 하지만, 상원을 비롯한 많은 고객이 조금 더 나은 기능을 원하고 있습니다. 주말에는 커피를 안 끓인다든지, 샤워 예정 시각 15분 전에 잔디밭 스프링클러를 자동으로 끈다든지, 쓰레기 분리수거를 하러 나가야 하는 날 아침에는 알람이 자동으로 더 빨리 울리도록 설정하는 기능이 있으면 좋겠죠?



해야 할 일: 객체가 할 일을 정리해야 합니다.

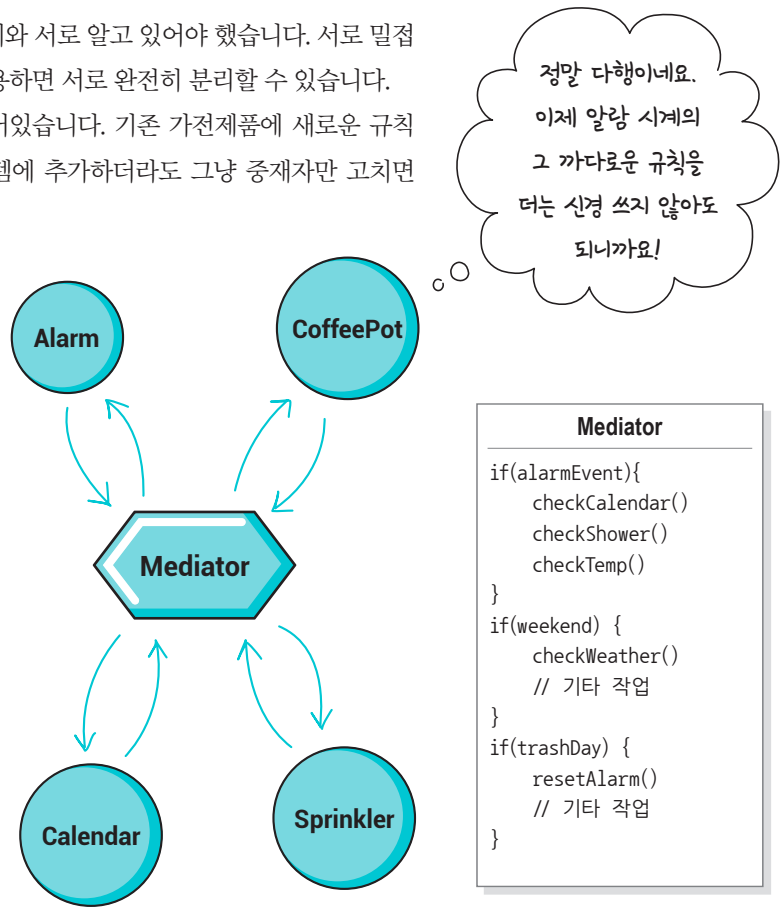
어떤 객체에 어떤 규칙을 넣어야 할지를 결정하기가 점점 어려워지고 있습니다. 그리고 여러 객체를 서로 연관시키는 과정도 점점 복잡해지고 있고요.

중재자 사용하기

시스템에 중재자 패턴을 적용하면 가전제품 객체들을 훨씬 단순화할 수 있습니다.

- 상태가 바뀔 때마다 중재자에게 알려 줍니다.
- 중재자에서 보낸 요청에 응답합니다.

중재자를 추가하기 전에는 모든 객체가 다른 객체와 서로 알고 있어야 했습니다. 서로 밀접하게 연관되어 있어야 하죠. 하지만 중재자를 사용하면 서로 완전히 분리할 수 있습니다. 중재자에는 모든 시스템을 제어하는 로직이 들어있습니다. 기존 가전제품에 새로운 규칙을 추가하거나 새로운 가전제품을 자동화 시스템에 추가하더라도 그냥 중재자만 고치면 됩니다.



중재자 패턴의 장점

- 시스템과 객체를 분리함으로써 재사용성을 획기적으로 향상시킬 수 있습니다.
- 제어 로직을 한 군데 모아놨으므로 관리하기가 수월합니다.
- 시스템에 들어있는 객체 사이에서 오가는 메시지를 확 줄이고 단순화할 수 있습니다.

중재자 패턴의 활용법과 단점

- 서로 연관된 GUI 구성 요소를 관리하는 용도로 많이 쓰입니다.
- 디자인을 잘 하지 못하면 중재자 객체가 너무 복잡해질 수 있다는 단점이 있습니다.

메멘토 패턴

객체를 이전의 상태로 복구해야 한다면 메멘토(Memento) 패턴을 쓰면 됩니다. 사용자가 '작업 취소'를 요청할 때를 생각해 보세요.

시나리오

최근에 발매한 롤플레잉 게임이 큰 성공을 거두었습니다. 이제 다들 그 게임에서 '13 레벨'까지 올릴 수 있게 되었죠. 하지만 점점 더 높은 레벨로 올라갈수록 캐릭터가 죽어서 게임이 끝날 확률이 높아집니다. 레벨을 올리려고 며칠씩 투자했는데 캐릭터가 죽어 버리면 정말 좌절에 빠지게 되죠. 처음부터 그 힘든 과정을 어떻게 다시 하겠습니까? 그러다 보니 캐릭터가 죽기 전에 저장해 뒀던 곳에서부터 다시 시작할 수 있게 세이브 기능을 추가해 달라는 요청이 빗발치게 되었습니다. 이 세이브 기능은 캐릭터가 죽기 전 마지막으로 레벨업했던 지점에서 다시 캐릭터를 살려서 플레이할 수 있게 해 주는 기능입니다.

메멘토 패턴 사용하기

메멘토 패턴에는 2가지 목적이 있습니다.

- 시스템에서 핵심적인 기능을 담당하는 객체의 상태 저장
- 핵심적인 객체의 캡슐화 유지

단일 역할 원칙을 떠올려 본다면, 저장하고자 하는 상태를 핵심 객체로부터 분리해 놓으면 좋겠다는 생각이 들 것입니다. 상태를 따로 저장하는 객체를 메멘토 객체라고 부릅니다.

메멘토 패턴의 장점

- 저장된 상태를 핵심 객체와는 다른 별도의 객체에 보관할 수 있어 안전합니다.
- 핵심 객체의 데이터를 계속해서 캡슐화된 상태로 유지할 수 있습니다.
- 복구 기능을 구현하기가 쉽습니다.

메멘토 패턴의 활용법과 단점

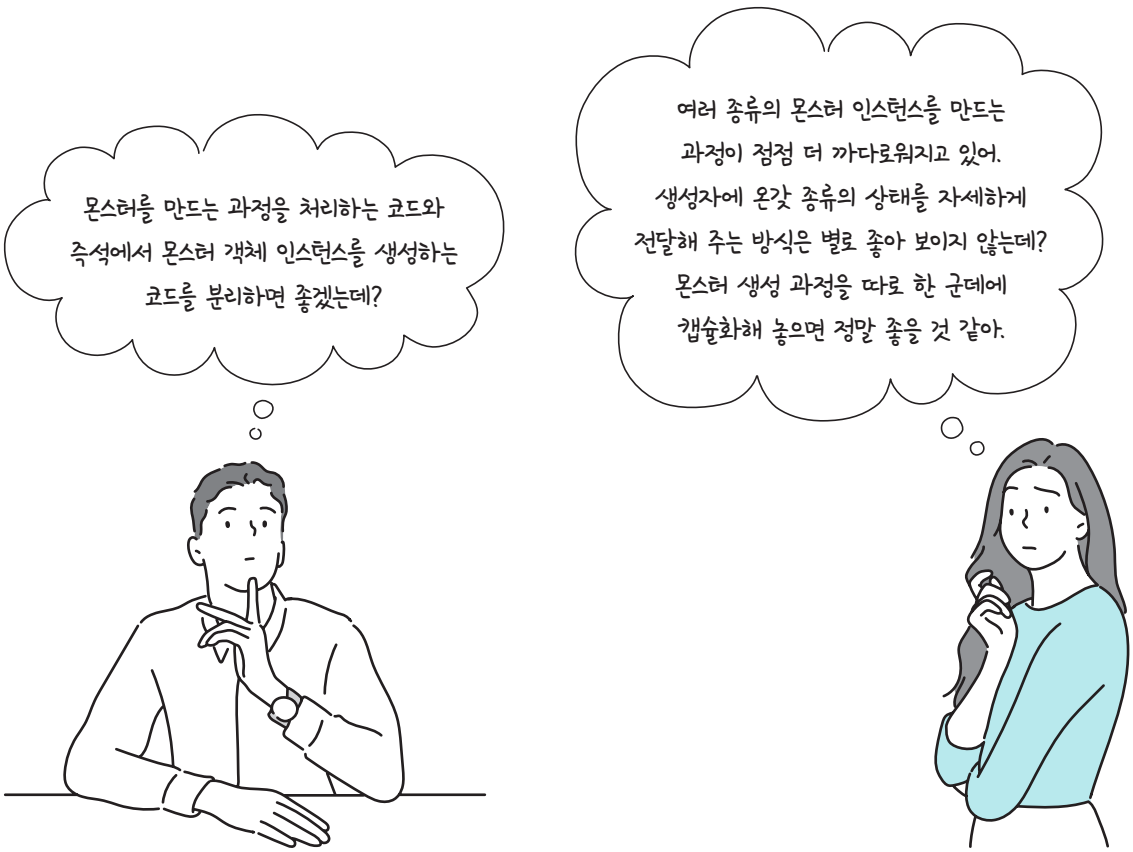
- 메멘토 객체를 써서 상태를 저장합니다.
- 자바 시스템에서는 시스템의 상태를 저장할 때 직렬화를 사용하는 것이 좋습니다.
- 상태를 저장하고 복구하는 데 시간이 오래 걸릴 수 있다는 단점이 있습니다.

프로토타입 패턴

어떤 클래스의 인스턴스를 만들 때 자원과 시간이 많이 들거나 복잡하다면 프로토타입(Prototype) 패턴을 쓰면 됩니다.

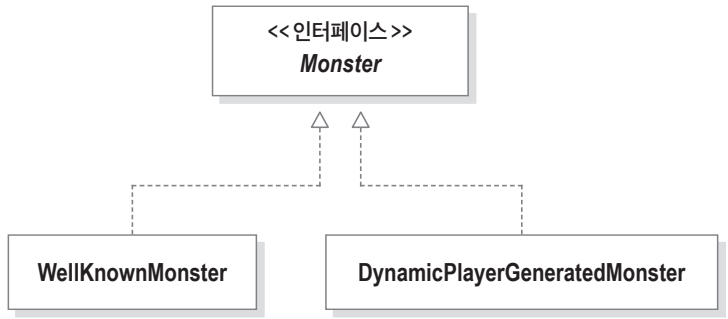
시나리오

우리가 만든 롤플레잉 게임에는 다양한 몬스터가 등장합니다. 히어로가 동적으로 생성되는 지형을 따라서 여행을 하면 끊임없이 몬스터가 등장하죠. 그런데 주변 환경에 맞춰서 몬스터의 특성이 바뀌면 좋겠습니다. 물속에 들어갔는데 새 모양의 몬스터가 캐릭터를 쫓아오면 정말 이상할 테니까요. 그리고 헤비 유저들이 직접 몬스터를 만들 수 있게 해 주려고 합니다.



프로토타입 패턴 사용하기

프로토타입 패턴을 사용하면 기존 인스턴스를 복사하기만 해도 새로운 인스턴스를 만들 수 있습니다(자바에서는 clone() 메소드를 사용하거나 역직렬화를 하면 되죠). 이 패턴의 가장 두드러진 특징은 클라이언트 코드에서 어떤 클래스의 인스턴스를 만드는지 전혀 모르는 상태에서도 새로운 인스턴스를 만들 수 있다는 점입니다.



MonsterMaker

```
makeRandomMonster() {
  Monster m =
    MonsterRegistry.getMonster();
}
```

클라이언트에서 현재 상황에 맞는 새로운 몬스터를 원합니다.
클라이언트는 몬스터의 종류를 알 수 없습니다.

MonsterMaker

```
Monster getMonster() {
  // 적당한 몬스터를 찾음
  return correctMonster.clone();
}
```

레지스트리에서 적당한 몬스터를 찾아서
클론을 만든 다음 그 클론을 리턴합니다.

프로토타입 패턴의 장점

- 클라이언트는 새로운 인스턴스를 만드는 과정을 몰라도 됩니다.
- 클라이언트는 구체적인 형식을 몰라도 객체를 생성할 수 있습니다.
- 상황에 따라서 객체를 새로 생성하는 것보다 객체를 복사하는 것이 더 효율적일 수 있습니다.

프로토타입 패턴의 활용법과 단점

- 시스템에서 복잡한 클래스 계층구조에 파묻혀 있는 다양한 형식의 객체 인스턴스를 새로 만들어야 할 때 유용하게 써먹을 수 있습니다.
- 때때로 객체의 복사본을 만드는 일이 매우 복잡할 수도 있다는 단점이 있습니다.

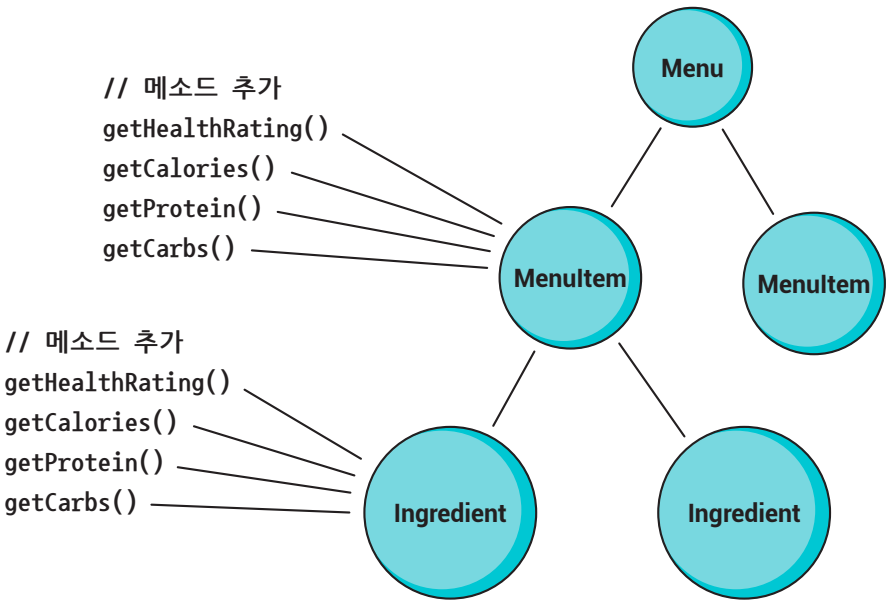
비지터 패턴

다양한 객체에 새로운 기능을 추가해야 하는데 캡슐화가 별로 중요하지 않다면 비지터(Visitor) 패턴을 쓰면 됩니다.

시나리오

객체마을 식당과 팬케이크 하우스의 단골손님들이 최근 들어 건강에 부쩍 관심을 가지게 되었습니다. 그래서 주문하기 전에 영양 정보를 요구하는 손님이 많이 늘었습니다. 그리고 어떤 손님은 재료별 영양 정보까지 요구하기도 합니다.

루가 제안한 방법

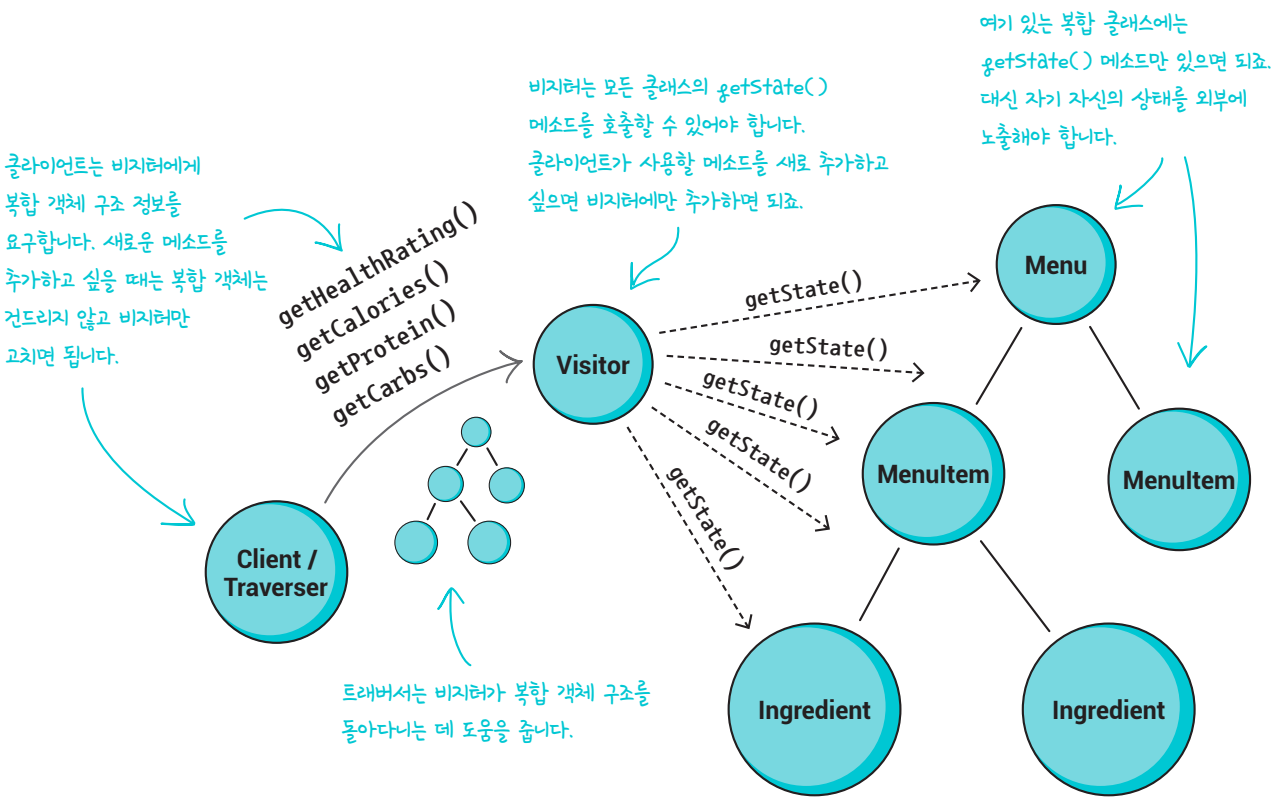


멜이 걱정하는 부분

"이건 판도라의 상자를 여는 것과 똑같다고… 나중에 메소드를 또 추가해야 할지 어떻게 알 수 있겠어? 게다가 매번 메소드를 추가할 때마다 두 군데에 코드를 추가해야 하잖아. 그리고 만약 조리법이 들어있는 클래스를 추가하는 식으로 기본 애플리케이션을 고쳐야 한다면 어떻게 해야 할까? 그러면 세 군데에서 코드를 고쳐야 하잖아! 너무 불편해"

비지터 패턴 사용하기

비지터 객체는 트래버서(Traverser) 객체와 함께 돌아갑니다. 트래버서는 컴포지트 패턴을 쓸 때, 복합 객체 내에 속해 있는 모든 객체에 접근하는 일을 도와주는 역할을 합니다. 비지터 객체에서 복합 객체 내의 모든 객체를 대상으로 원하는 작업을 처리하게 해 주는 거죠. 각각의 상태를 모두 가져오면 클라이언트는 비지터에게 각 상태에 맞는 다양한 작업을 처리하도록 요구할 수 있습니다. 새로운 기능이 필요하게 되더라도 비지터만 고치면 되니까 편리하죠.



비지터 패턴의 장점

- 구조를 변경하지 않으면서도 복합 객체 구조에 새로운 기능을 추가할 수 있습니다.
- 비교적 손쉽게 새로운 기능을 추가할 수 있습니다.
- 비지터가 수행하는 기능과 관련된 코드를 한곳에 모아 둘 수 있습니다.

비지터 패턴의 단점

- 비지터를 사용하면 복합 클래스의 캡슐화가 깨집니다.
- 컬렉션 내의 모든 항목에 접근하는 트래버서가 있으므로 복합 구조를 변경하기가 더 어려워집니다.