자바 매거진

따르다:

Java 및 JVM 개발자를 위해 Java 커뮤니티에서 작성

코딩 , 디자인 패턴

Java에서 Joshua Bloch의 Builder 디자인 패턴 탐색

2021년 5월 29일 | 12분 읽기

프랭크 키위

블록의 빌더 패턴은 누락된 언어 기능에 대한 해결책으로 생각할 수 있습니다.

이 기사의 PDF를 다운로드하세요

에리히 감마 등이 설명한 23개의 Gang of Four(GoF) 디자인 패턴 중 하나인 빌더 패턴은 복잡한 객체를 단계별로 구성할 수 있는 생성적 디자인 패턴입니다. 동일한 구성 코드를 사용하여 다양한 유형과 제품 표현을 생성할 수 있습니다. 그러나 이 패턴은 동일한 빌드 프로세스를 사용하여 다양한 불변 객체를 빌드해야 하는 경우에만 사용해야 합니다.

빌더 패턴은 또 다른 중요한 GoF 생성 패턴인 추상 팩토리 패턴과 크게 다르지 않습니다. 빌더 패턴이 복잡한 객체를 단계별로 구성하는 데 초점을 맞추는 반면, 추상 팩토리 패턴은 `Product`간단하거나 복잡한 객체 패밀리를 강조합니다. 빌더 패턴이 마지막 단계로 final을 반환하는 반면 `Product`, 추상 팩토리 패턴은 `Product`즉시 반환합니다.

디자인 패턴은 언어와 무관하지만, 각 언어의 기능에 따라 구현 방식이 다르기 때문에 이 글의 마지막 섹션에서 보여드리겠지만 일부 패턴은 불필요하기도 합니다.

이 글에서는 Joshua Bloch의 Builder 패턴 버전(Effective Java의 Builder 패턴이라고도 함, 그의 책에서 이름을 따옴)에 초점을 맞춥니다 . *이 패턴* 버전 은 GoF Builder 패턴의 변형이며 종종 혼동됩니다.

블로흐의 Builder 패턴 버전은 많은 선택적 매개변수가 있는 객체를 빌드하는 간단하고 안전한 방법을 제공하므로, 텔레스코핑 생성자 문제(곧 설명하겠습니다)를 해결합니다. 또한, 대부분의 경우 동일한 유형의 매개변수가 여러 개 있는 대형 생성자의 경우, 어떤 값이 어떤 매개변수에 속하는지 항상 명확하지 않습니다. 따라서 매개변수 값을 섞을 가능성이 높습니다.

블로흐의 빌더 패턴에서 사용하는 관용구는 `Builder`외부 클래스(빌드되는 제품)의 인스턴스를 생성하지 않고도 액세스할 수 있는 정적 내부 클래스를 생성하여 이러한 문제를 해결하지만 여전히 외부 개인 생성자에는 액세스할 수 있습니다.

명확하게 하기 위해 앞으로 *빌더 패턴이라는 용어를 사용할 때는 특별히 다른 언급이 없는 한 블로흐 버전의 빌더 패턴을 의미합니다.*

더 깊이 들어가기 전에, 이 문서 전체에서 다음 예제 클래스를 사용하겠습니다.

코드 조각 복사

```
import builder.pattern.Genre;
import java.time.Year;

public class Book {
    private final String isbn;
    private final String title;
    private final Genre genre;
    private final String author;
    private final Year published;
    private final String description;
    public Book(String isbn, String title, Genre genre, String author, Year published, String description) {
        this.isbn = isbn;
        this.title = title;
        this.genre = genre;
        this.author = author;
        this.published = published;
        this.description = description;
    }

    public String getIsbn() {
        return isbn;

    }
    public String getTitle() {

        return title;
    }

    public Genre getGenre() {
        return genre;
    }

    public String getAuthor() {
        return author;
    }

    public Year getPublished() {
        return published;
```

```
    }

    public String getDescription() {
        return description;
    }

}
```

클래스 `Book`에는 6개의 최종 필드가 있는데, 모든 매개변수를 설정하는 생성자 하나와 객체가 생성되면 객체의 필드를 읽는 해당 게터가 있습니다. 결과적으로 이 클래스에서 파생된 모든 객체는 변경 불가능합니다.

또한, 이 `Book`클래스에는 두 개의 필수 필드가 있습니다. ISBN은 책의 10자리 또는 13자리 국제 표준 도서 번호를 나타냅니다. `Title`나머지 필드는 모두 선택 사항입니다.

이제 질문이 생깁니다. 각 조합에 적합한 생성자를 사용하여 다양한 옵션 매개변수 조합으로 객체를 어떻게 구성할 수 있을까요? 객체는 불변하도록 의도되었기 때문에 Enterprise JavaBean과 유사한 세터는 불가능합니다.

망원경 건설자

가능한 해결책 중 하나는 텔레스코픽 생성자 로 , 첫 번째 생성자는 필수 필드만 취합니다. 모든 선택적 필드에는 필수 필드 *와* 선택적 필드를 모두 취하는 추가 생성자가 있습니다. 모든 생성자는 `null`누락된 매개변수 대신 값을 전달하여 후속 생성자를 호출합니다. 체인의 마지막 생성자만 매개변수에서 제공하는 값을 사용하여 모든 필드를 설정합니다.

아래에서는 `Book`텔레스코픽 생성자 솔루션이 있는 클래스를 볼 수 있습니다.

코드 조각 복사

```
import builder.pattern.Genre;
import java.time.Year;

public class Book {
    private final String isbn;
    private final String title;
    private final Genre genre;
    private final String author;
    private final Year published;
    private final String description;

    public Book(String isbn, String title) {
        this(isbn, title, null);
    }
```

```
public Book(String isbn, String title, Genre genre) {
    this(isbn, title, genre, null);
}

public Book(String isbn, String title, Genre genre, String author) {
    this(isbn, title, genre, author, null);
}

public Book(String isbn, String title, Genre genre, String author, Year published) {
    this(isbn, title, genre, author, published, null);
}

public Book(String isbn, String title, Genre genre, String author, Year published, String description) {
    this.isbn = isbn;
    this.title = title;
    this.genre = genre;
    this.author = author;
    this.published = published;
    this.description = description;
}

public String getIsbn() {
    return isbn;
}

public String getTitle() {
    return title;
}

public Genre getGenre() {
    return genre;
}

public String getAuthor() {
    return author;
}

public Year getPublished() {
    return published;
}

public String getDescription() {
    return description;
}
```

```
    }

}
```

안타깝게도, 텔레스코핑 생성자는 어떤 경우에는 값을 전달해야 하는 것을 막지 못합니다 . 예를 들어 , , , 를 사용 `null`하여 를 만들어야 한다면 어떻게 하시겠습니까? 그런 생성자는 없습니다!`BookISBNtitleauthor`

아마도 기존 생성자를 사용하고 `null`누락된 매개변수 대신 값을 전달할 것입니다.

코드 조각 복사

```
new Book("0-12-345678-9", "Moby-Dick", null, "Herman Melville");
```

`null`그러나 다음과 같이 적절한 생성자를 만들면 값 사용을 피할 수 있습니다.

코드 조각 복사

```
public Book(String isbn, String title, String author) {
    this.isbn = isbn;
    this.title = title;
    this.author = author;
}
```

결과적으로 생성자 호출은 잘 작동해야 하지만 다른 문제가 발생할 수 있습니다.

코드 조각 복사

```
new Book("0-12-345678-9", "Moby-Dick", "Herman Melville");
```

또한 . 대신 `Book`with `ISBN`and `title`but with를 만들어야 한다고 상상해 보세요 . 다음과 같은 생성자를 추가하고 싶을 수도 있습니다.`descriptionauthor`

코드 조각 복사

```
public Book(String isbn, String title, String description) {
    this.isbn = isbn;
    this.title = title;
    this.description = description;
}
```

```
}
```

이것은 작동하지 않을 것입니다. 동일한 서명의 두 생성자는 컴파일러가 어느 것을 선택해야 할지 알 수 없기 때문에 동일한 클래스에 공존할 수 없습니다. 또한, 모든 유용한 매개변수 조합에 대해 생성자를 만들면 생성자의 조합이 커져서 결과 코드를 읽기 어렵고 유지 관리하기가 더 어려워집니다.

따라서 다수의 선택적 필드가 있는 객체의 생성과 관련된 문제를 해결하기 위해 텔레스코픽 생성자나 그 외의 생성자 매개변수의 가능한 조합은 좋은 방법이 아닙니다.

여기서 Bloch의 Builder 패턴 버전이 등장합니다.

효과적인 Java 빌더 패턴

앞서 언급했듯이, Bloch의 Builder 패턴은 GoF Builder 패턴의 변형입니다.

GoF Builder 패턴에는 `Director`, `Builder`(인터페이스), `ConcreteBuilder`(구현) 및 의 네 가지 구성 요소가 있습니다 `Product`. 여기서는 개별 구성 요소에 대해 다루지 않겠습니다. 이 글의 범위를 벗어나기 때문입니다.

`ConcreteBuilder`블록의 빌더 패턴은 GoF의 대응 패턴의 약어로, 4가지 구성 요소 중 와 로만 구성된다는 의미입니다 . 또한 블록의 빌더는 중첩된 정적 클래스(클래스 자체 내부에 위치)로 구성되므로 `Product`Java에 특화된 구현을 갖습니다 .`BuilderProduct`

사실, 이 관용어구는 객체 지향적 디자인 패턴이 아니라 명명된 매개변수가 부족하다는 언어적 특징의 부족을 해결하기 위한 해결책입니다.

어떻게 작동하나요?

`Builder`먼저 필수 필드를 생성자에 전달하여 클래스 인스턴스를 만듭니다 . 그런 다음 `Builder`클래스의 세터와 유사한 메서드를 호출하여 선택적 필드에 대한 값을 설정합니다. 모든 필드를 설정한 후 인스턴스 `build`에서 메서드를 호출합니다. 이 메서드는 이전에 설정된 값을 생성자에 전달하여 `Builder`를 만들고 결국 새 인스턴스를 반환합니다.`ProductProductProduct`

구현 방법은 다음과 같습니다.

코드 조각 복사

```
import builder.pattern.Genre;
import java.time.Year;

public class Book {
    private final String isbn;
    private final String title;
    private final Genre genre;
    private final String author;
    private final Year published;
    private final String description;
    private Book(Builder builder) {
```

```
        this.isbn = builder.isbn;
        this.title = builder.title;
        this.genre = builder.genre;
        this.author = builder.author;
        this.published = builder.published;
        this.description = builder.description;
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public Genre getGenre() {
        return genre;
    }

    public String getAuthor() {
        return author;
    }

    public Year getPublished() {
        return published;
    }

    public String getDescription() {
        return description;
    }

    public static class Builder {
        private final String isbn;
        private final String title;
        private Genre genre;
        private String author;
        private Year published;
        private String description;

        public Builder(String isbn, String title) {
            this.isbn = isbn;
            this.title = title;
        `
```

```
        }

        public Builder genre(Genre genre) {
            this.genre = genre;
            return this;
        }

        public Builder author(String author) {
            this.author = author;
            return this;
        }

        public Builder published(Year published) {
            this.published = published;
            return this;
        }

        public Builder description(String description) {
            this.description = description;
            return this;
        }

        public Book build() {
            return new Book(this);
        }

    }

}
```

다음은 유의해야 할 몇 가지 사항입니다.

생성자 의 범위가 Book로 변경되어 private클래스 외부에서 액세스할 수 없습니다 Book. 이로 인해 인스턴스를 직접 생성할 수 없습니다 Book. 객체 생성 프로세스는 Builder클래스에 위임됩니다.

생성자 는 생성자가 설정할 모든 값을 포함하는 인스턴스를 유일한 매개변수로 Book사용합니다 . 또는 생성자는 필드에 해당하는 모든 매개변수를 사용할 수 있지만 , 이렇게 하면 's 메서드 에서 생성자를 호출할 때 올바른 순서로 설정해야 하는 많은 매개변수를 다시 처리해야 합니다 . 동일한 유형의 매개변수를 섞는 것은 개발자가 패턴을 구현하여 피하려고 하는 잠재적인 문제 중 하나입니다 .BuilderBookBookBookBookBuilderbuildBuilder

클래스 는 생성자에게 전달될 값을 보관하는 데 필요한 클래스 Builder와 동일한 필드를 포함합니다 . 이는 종종 코드 중복으로 올바르게 비판을 받았습니다.BookBook

설정할 모든 선택적 필드에 대해 Builder클래스는 필드의 값을 할당하고 현재 Builder인스턴스를 반환하여 객체를 유창한 방식으로 빌드하는 세터와 유사한 메서드를 노출합니다. 각 메서드 호출은 동일한 Builder인스턴스를 반환하므로 메서드 호출을 체인으로 연결할 수 있어 클라이언트 코드가 더 간결하고 읽기 쉽습

니다.

이 메서드는 현재 인스턴스를 유일한 매개변수로 전달하여 생성자를 `build`호출합니다 . 인스턴스가 보유한 값은 생성자에 의해 압축 해제되어 해당 필드에 할당됩니다.`BookBuilderBuilderBookBook`

이렇게 `Builder`사용됩니다.

코드 조각 복사

```
Book book = new Book.Builder("0-12-345678-9", "Moby-Dick")
                .genre(Genre.ADVENTURE_FICTION)
                .author("Herman Melville")
                .published(Year.of(1851))
                .description(
                        "The book is the sailor Ishmael's narrative of the obsessive quest of "
                        + "Ahab, captain of the whaling ship Pequod, for revenge on Moby Dick, "
                        + "the giant white sperm whale that on the ship's previous voyage bit "
                        + "off Ahab's leg at the knee."
                )
                .build();
```

재사용성 및 제한 사항

Builder 패턴은 또한 `Builder`이전 생성 프로세스에서 이미 채워진 기존 인스턴스를 재사용할 수 있도록 합니다. 이를 통해 모든 값을 다시 설정할 필요가 없으므로 몇 가지 다른 속성 값만 있는 새 객체를 쉽게 만들 수 있습니다.

이것이 예를 들어 어떻게 작동하는지 살펴보겠습니다 `Book`. 허먼 멜빌의 *모비딕은* 여러 판으로 출판되었습니다. 첫 번째 판은 1851년에 출판되었습니다. 1952년에 출판된 두 번째 판에는 25페이지 분량의 서문과 250페이지가 넘는 설명 노트가 포함되었습니다.

`Book`1952년판에 대한 새 객체를 만들고 싶다면 , 이전에 만든 `Builder`1851년판 인스턴스를 재사용하고 출판 날짜를 재정의한 다음 메서드를 다시 호출하여 1952년판에 해당하는 `build`새 객체를 생성하면 됩니다 .`Book`

코드 조각 복사

```
Book.Builder bookBuilder = new Book.Builder("0-12-345678-9", "Moby-Dick")
                .genre(Genre.ADVENTURE_FICTION)
                .author("Herman Melville")
                .published(Year.of(1851))
                .description("description omitted for brevity");

// Create a first Book object
Book book = bookBuilder.build();
```

```
// Create a second, slightly different, object reusing the same Builder instance
book = bookBuilder.published(Year.of(1952)).build();
```

그러나 위의 예는 그다지 현실적이지 않습니다. ISBN도 변경해야 하기 때문입니다. ISBN 필드는 '의 생성자를 `final`통해 설정되어야 하기 때문에 불가능합니다 `Builder`. 그러면 새 인스턴스가 생성됩니다 `Book`. 이 예는 '의 재사용성의 한계를 보여줍니다 `Builder`.

상태 검증

Bloch의 Builder 패턴은 인스턴스의 생성 프로세스 중에 편리한 상태 검증도 허용합니다 `Product`. 모든 `Book`필드가 이므로 인스턴스가 생성된 `final`후에는 변경할 수 없으므로 상태는 특히 생성 시점에 한 번만 검증하면 됩니다. 검증 로직은 's 메서드 또는 생성자 `Book`에서 구현(또는 호출)할 수 있습니다 . 다음 예에서 로직은 메서드에서 호출됩니다.`Builder``build``Book`

코드 조각 복사

```
public static class Builder {
    private final IsbnValidator isbnValidator = new IsbnValidator();
    private final String isbn;
    private final String title;
    private Genre genre;
    private String author;
    private Year published;
    private String description;

    public Builder(String isbn, String title) {
        this.isbn = isbn;
        this.title = title;
    }

    public Builder genre(Genre genre) {
        this.genre = genre;
        return this;
    }

    public Builder author(String author) {
        this.author = author;
        return this;
    }

    public Builder published(Year published) {
        this.published = published;
        return this;
```

```
        return this;
    }

    public Builder description(String description) {
        this.description = description;
        return this;
    }

    public Book build() throws IllegalStateException {
        validate();
        return new Book(this);
    }

    private void validate() throws IllegalStateException {
        MessageBuilder mb = new MessageBuilder();
        if (isbn == null) {
            mb.append("ISBN must not be null.");
        } else if (!isbnValidator.isValid(isbn)) {
            mb.append("Invalid ISBN!");
        }
        if (title == null) {
            mb.append("Title must not be null.");
        } else if (title.length() < 2) {
            mb.append("Title must have at least 2 characters.");
        } else if (title.length() > 100) {
            mb.append("Title cannot have more than 100 characters.");
        }
        if (author != null && author.length() > 50) {
            mb.append("Author cannot have more than 50 characters.");
        }
        if (published != null && published.isAfter(Year.now())) {
            mb.append("Year published cannot be greater than current year.");
        }
        if (description != null && description.length() > 500) {
            mb.append("Description cannot have more than 500 characters.");
        }
        if (mb.length() > 0) {
            throw new IllegalStateException(mb.toString());
        }
    }

}
```

실제 객체가 생성되기 *전에* 유효성 검사 논리를 호출하면 해당 `Book`객체에 의해 생성된 모든 인스턴스가 `Builder`유효한 상태를 갖는다는 것을 보장할 수 있습니다.

자바 레코드

이전 기사인 " Java 레코드 탐구: 직렬화, 마샬링 및 빈 상태 검증 "에는 Java 레코드에 구현된 Bloch의 Builder 패턴의 예가 포함되었습니다. 실제로 레코드는 본질적으로 불변 구조이기 때문에 Bloch의 Builder 구현에 적합합니다.

코드 조각 복사

```
public record BookRecord(String isbn, String title, Genre genre, String author, Year published, String description) {

    private BookRecord(Builder builder) {
        this(builder.isbn, builder.title, builder.genre, builder.author, builder.published, builder.description);
    }

    public static class Builder {
        private final IsbnValidator isbnValidator = new IsbnValidator();
        private final String isbn;
        private final String title;
        private Genre genre;
        private String author;
        private Year published;
        private String description;

        public Builder(String isbn, String title) {
            this.isbn = isbn;
            this.title = title;
        }

        public Builder genre(Genre genre) {
            this.genre = genre;
            return this;
        }

        public Builder author(String author) {
            this.author = author;
            return this;
        }

        public Builder published(Year published) {
            this.published = published;
            return this;
        }
```

```
        public Builder description(String description) {
            this.description = description;
            return this;
        }

        public BookRecord build() throws IllegalStateException {
            validate();
            return new BookRecord(this);
        }

        private void validate() throws IllegalStateException {
            MessageBuilder mb = new MessageBuilder();
            if (isbn == null) {
                mb.append("ISBN must not be null.");
            } else if (!isbnValidator.isValid(isbn)) {
                mb.append("Invalid ISBN!");
            }
            if (title == null) {
                mb.append("Title must not be null.");
            } else if (title.length() < 2) {
                mb.append("Title must have at least 2 characters.");
            } else if (title.length() > 100) {
                mb.append("Title cannot have more than 100 characters.");
            }
            if (author != null && author.length() > 50) {
                mb.append("Author cannot have more than 50 characters.");
            }
            if (published != null && published.isAfter(Year.now())) {
                mb.append("Year published cannot be greater than current year.");
            }
            if (description != null && description.length() > 500) {
                mb.append("Description cannot have more than 500 characters.");
            }
            if (mb.length() > 0) {
                throw new IllegalStateException(mb.toString());
            }
        }

    }

}
```

The example above uses an alternative constructor to pass the `Builder` instance to the `record` constructor. In an alternative constructor, the canonical constructor (the one generated by the compiler) must be called *before* you can add any further statements. This means that the values of the `Builder` fields must be passed to the constructor parameters and, therefore, cannot be assigned directly to the `record` fields.

There's another choice: You can call the canonical constructor directly from the `Builder`'s `build` method. Either way, ensure that the constructor parameters are not mixed up.

Fortunately, with records, you do not have any code duplication as you have with regular classes, because the compiler generates the record fields and accessors. You can declare the fields only once explicitly in the `Builder` class.

Named parameters

If Java had named parameters, Bloch's version of the Builder pattern would be unnecessary, because you could provide only those parameters currently needed to create the object. Look at the following constructor:

Copy code snippet

```
public Book(String isbn = null, String title = null, Genre genre = Genre.UNKNOWN, String author = null, Year publishe
    this.isbn = isbn;
    this.title = title;
    this.genre = genre;
    this.author = author;
    this.published = published;
    this.description = description;
}
```

Below are two examples of how the constructor can be called.

Copy code snippet

```
new Book(isbn = "0-12-345678-9", title = "Moby-Dick", author = "Herman Melville");
new Book(isbn = "0-12-345678-9", title = "Moby-Dick", published = Year.of(1952), author = "Herman Melville");
```

With named parameters, you need to define only a single constructor that works for all possible combinations of parameters. Thus, the number of parameters used and the order in which they are set does not matter. The omitted parameters take the default values specified in the constructor definition.

Conclusion

With Bloch's version of the Builder pattern, you can create objects that have many optional parameters without using cumbersome and error-prone telescoping

With Bloch's version of the Builder pattern, you can create objects that have many optional parameters without using cumbersome and error-prone telescoping constructors. Further, the pattern avoids mixing up parameter values in large constructors that often have multiple consecutive parameters of the same type.

In addition, the same `Builder` instance can be used to create other objects of the same type that have slightly different attribute values than the one created in the first construction process.

The Builder pattern also allows for easy state validation by implementing or calling the validation logic in the `build` method, before the actual object is created. This avoids the creation of objects with invalid state.

When the pattern is used with records, there is no code duplication as is the case with regular classes, which require the same fields to be specified in the `Product` and `Builder` classes.

Finally, if Java had named parameters, Bloch's version of the Builder pattern would be superfluous.

Dig deeper

- Diving into Java records: Serialization, marshaling, and bean state validation
- Review of Joshua Bloch's *Effective Java* (third edition)
- Interview of Google's Joshua Bloch

Frank Kiwy

Frank Kiwy는 유럽의 정부 IT 센터에서 일하는 선임 소프트웨어 개발자이자 프로젝트 리더입니다. 그의 초점은 Java SE, Java EE 및 웹 기술입니다. Kiwy는 또한 소프트웨어 아키텍처에 관심이 있으며 지속적인 통합 및 제공에 전념합니다. 그는 현재 여러 프로젝트를 담당하고 있는 유럽 연합의 공통 농업 정책을 구현하는 데 참여하고 있습니다. 프로그래밍할 때 그는 명확하고 이해하기 쉬운 API가 있는 잘 설계된 소프트웨어를 중시합니다.
더 보기

이전 게시물

Jakarta EE 애플리케이션을 실행하려면 애플리케이션 서버가 필요하지 않습니다.

Arjan Tijms | 15 분 읽기

다음 게시물

스스로 퀴즈를 풀어보세요: Java 객체는 언제 가비지 수집의 대상이 될까요?

미칼라이 자이킨 | 6 분 읽기