C# Design Patterns: Bridge

THE BRIDGE DESIGN PATTERN

Vladimir Khorikov

@vkhorikov www.enterprisecraftsmanship.com

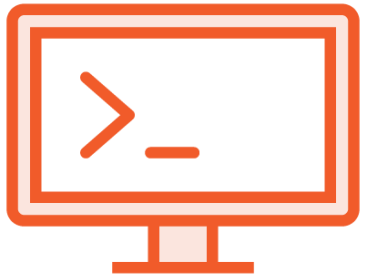
The Bridge Design Pattern

Its purpose is to decouple an abstraction from its implementation so that the two can vary independently.

The Bridge Design Pattern

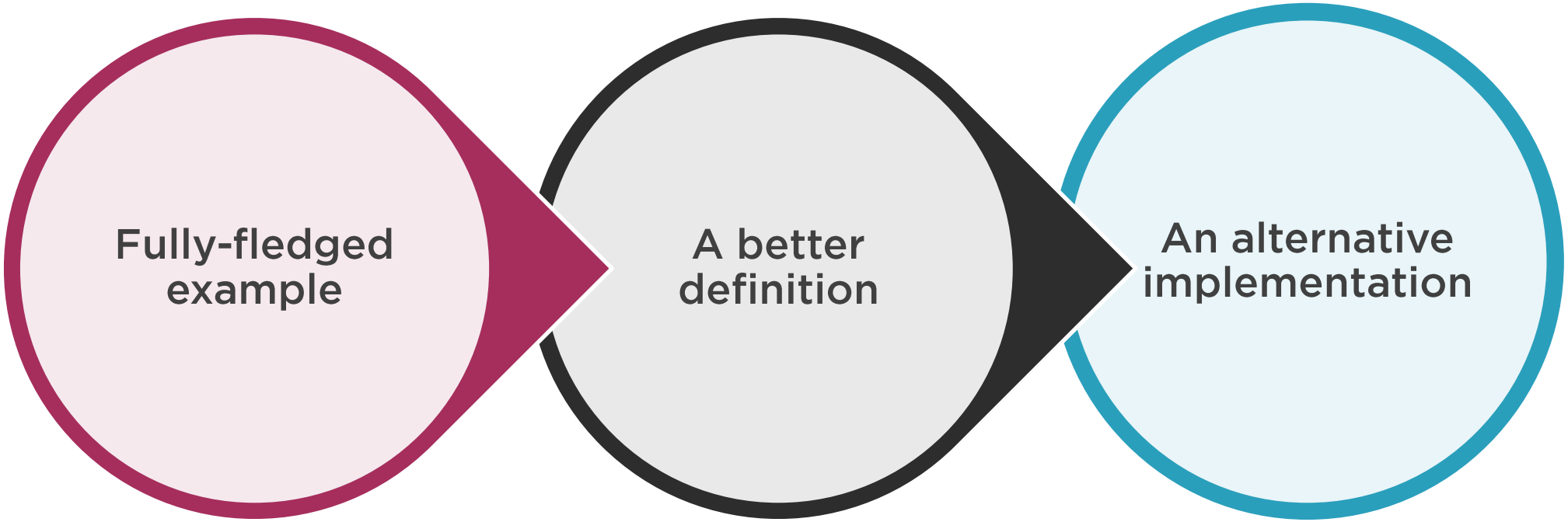
Its purpose is to decouple an abstraction from its implementation so that the two can vary independently.

The Bridge Design Pattern



Hard to relate to examples about desktop UI

The Bridge Design Pattern



The Bridge Design Pattern

http://bit.ly/bridge-pattern

The Bridge Design Pattern

vkhorikov / BridgePattern

Unwatch 2 | Star 1 | Fork 0

Code | Issues | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

master

Commits on Aug 20, 2020

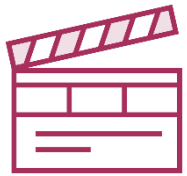
Initial

vkhorikov committed 7 days ago

885d094

Newer | Older

Sample Application Introduction



Online movie theater

MovieLicense

Movie
PurchaseTime
GetPrice()
GetExpirationDate()

TwoDaysLicense

LifeLongLicense

Sample Application Introduction

What challenges the bridge pattern helps to overcome?

- New requirements
- Using a naïve approach
- Refactoring using the bridge pattern

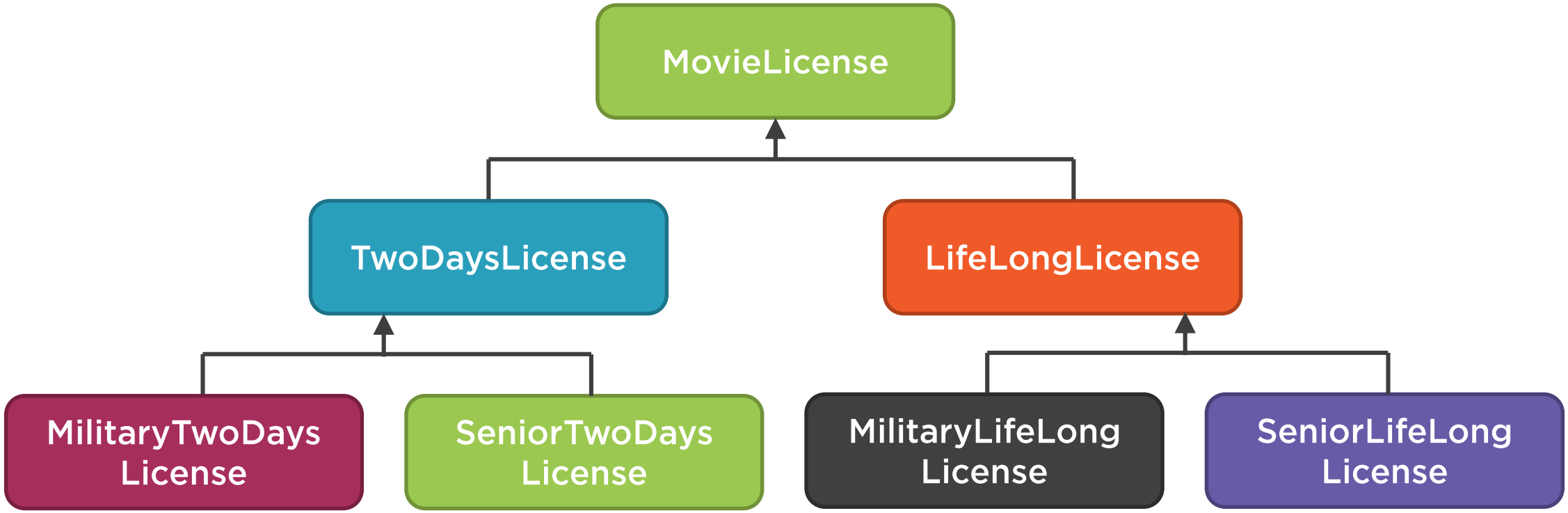
A New Requirement: Discounts

Discounts

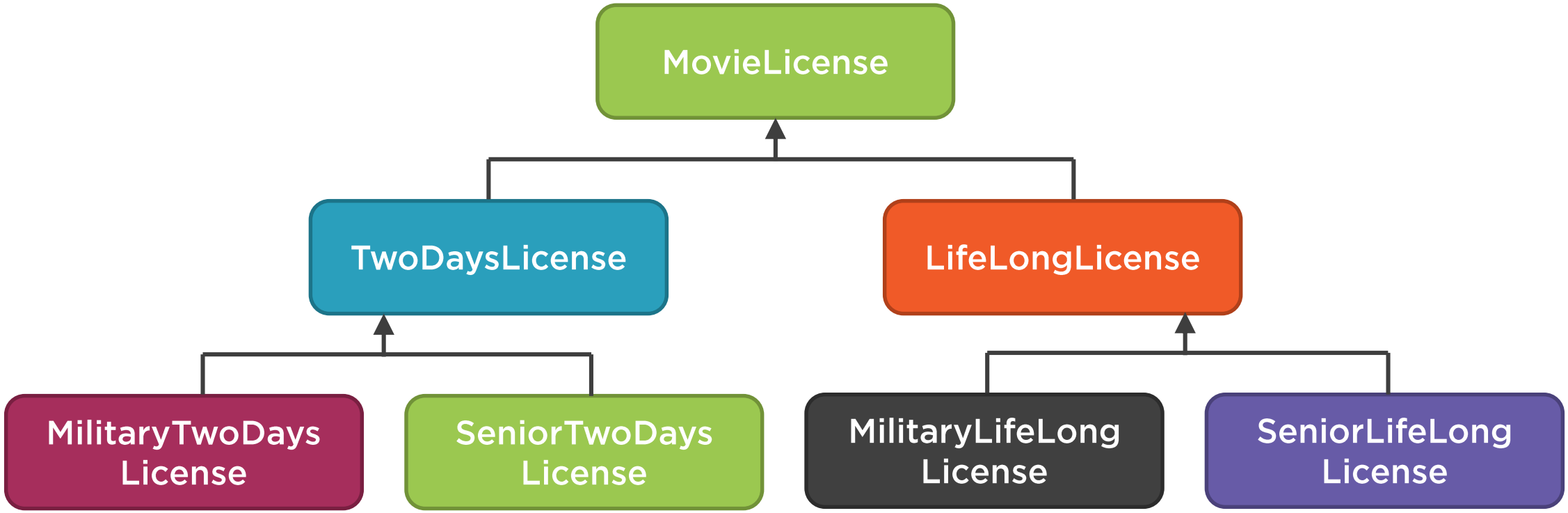
10% military discount

20% senior discount

A New Requirement: Discounts



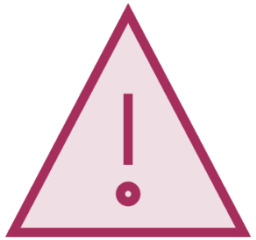
Recap: The Naive Approach



Exponential growth of complexity

2 → 6 → 12 → 18

Recap: The Naive Approach

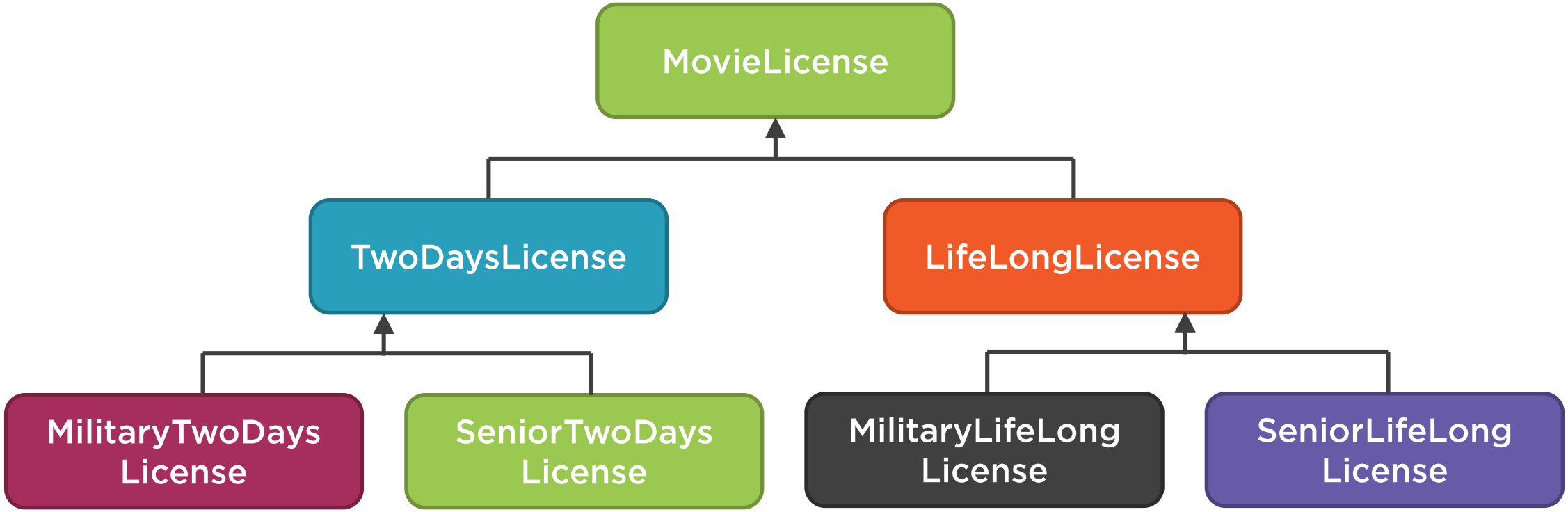


Domain knowledge duplication

```
public class MilitaryTwoDaysLicense
    : TwoDaysLicense
{
    public override decimal GetPrice()
    {
        return base.GetPrice() * 0.9m;
    }
}
public class MilitaryLifeLongLicense
    : LifeLongLicense
{
    public override decimal GetPrice()
    {
        return base.GetPrice() * 0.9m;
    }
}
```

```
public class SeniorTwoDaysLicense
    : TwoDaysLicense
{
    public override decimal GetPrice()
    {
        return base.GetPrice() * 0.8m;
    }
}
public class SeniorLifeLongLicense
    : LifeLongLicense
{
    public override decimal GetPrice()
    {
        return base.GetPrice() * 0.8m;
    }
}
```

Recap: The Naive Approach

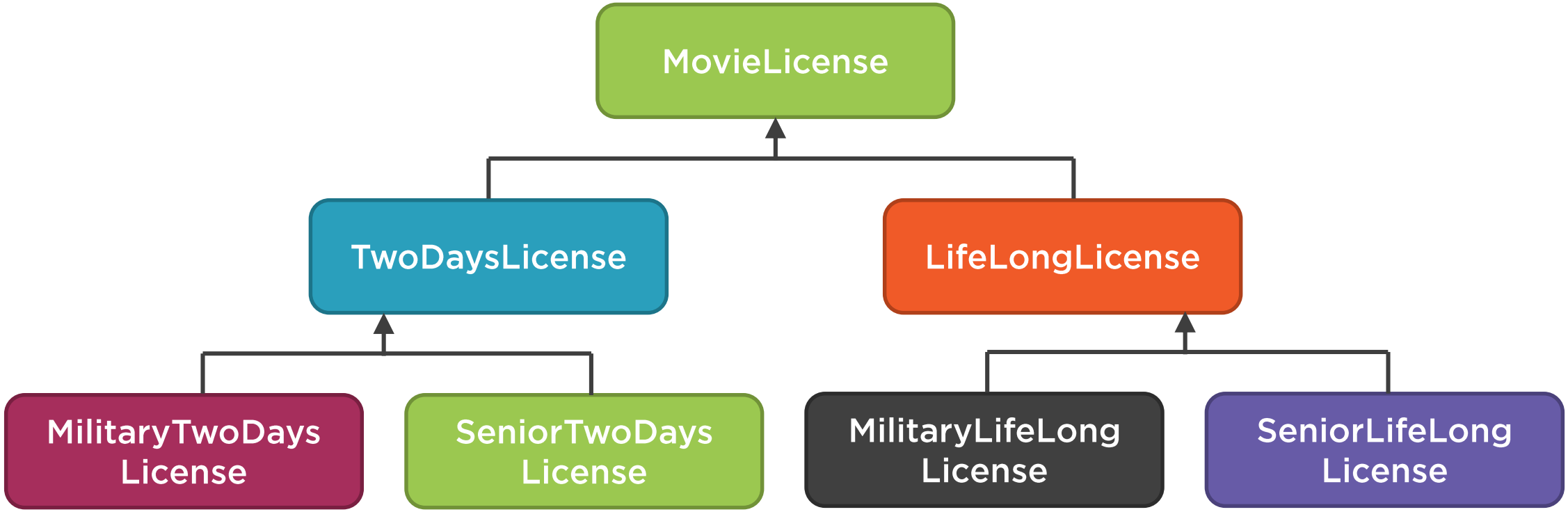


Split the hierarchy

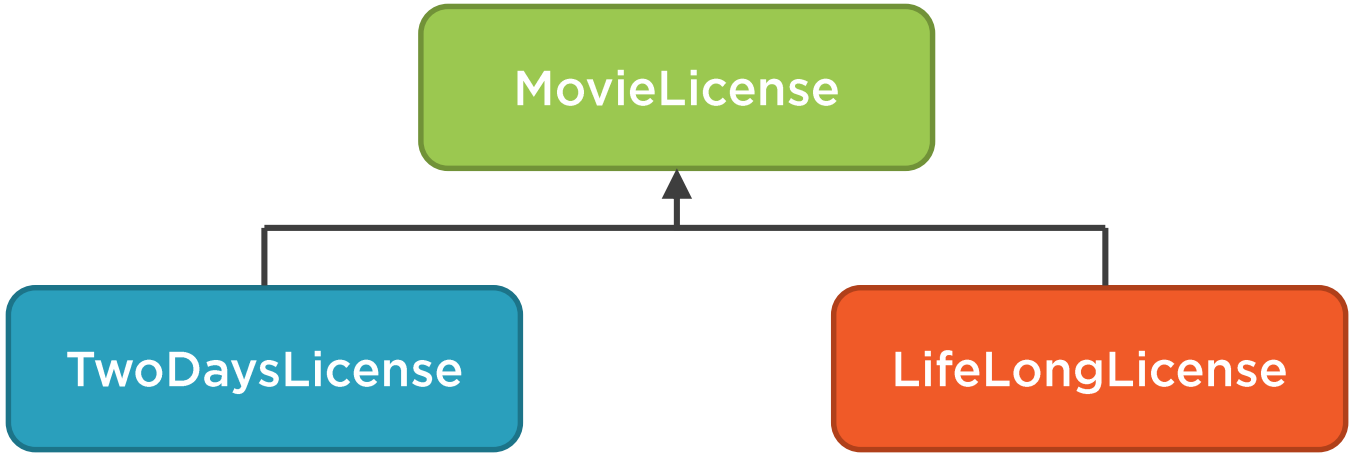
Applying the Bridge Pattern

Split the class hierarchy

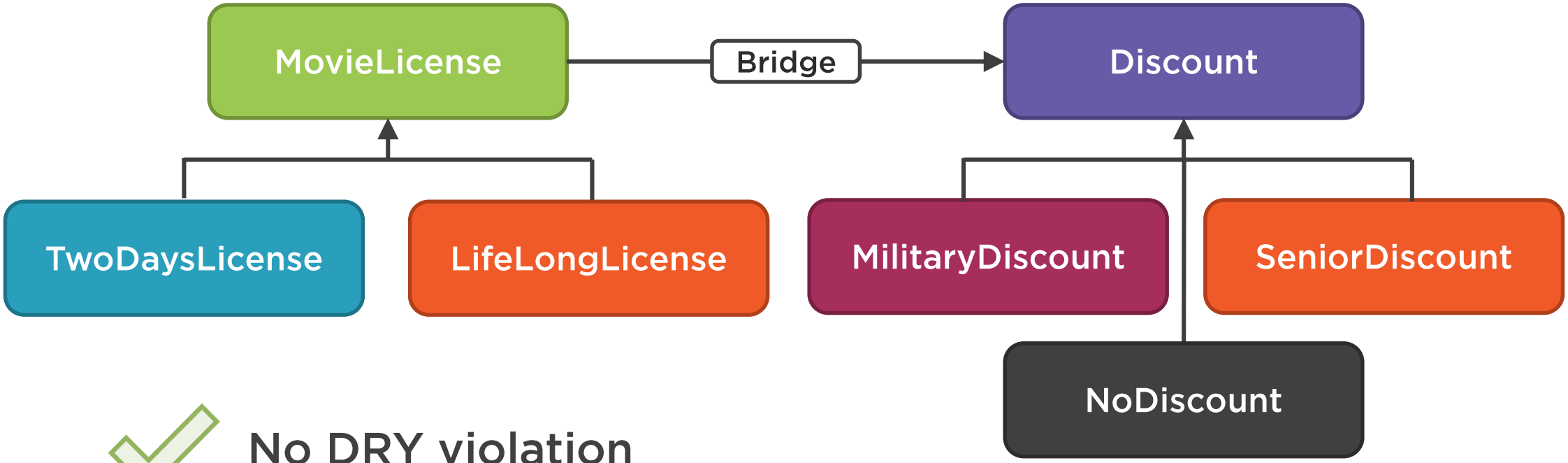
Recap: Applying the Bridge Pattern



Recap: Applying the Bridge Pattern



Recap: Applying the Bridge Pattern



- No DRY violation
- Simplified the code

2 licenses x 3 discounts = 6 classes

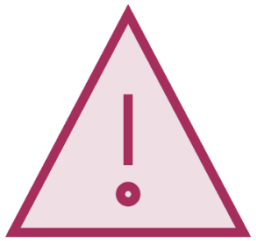
5 licenses x 5 discounts = 25 classes

vs

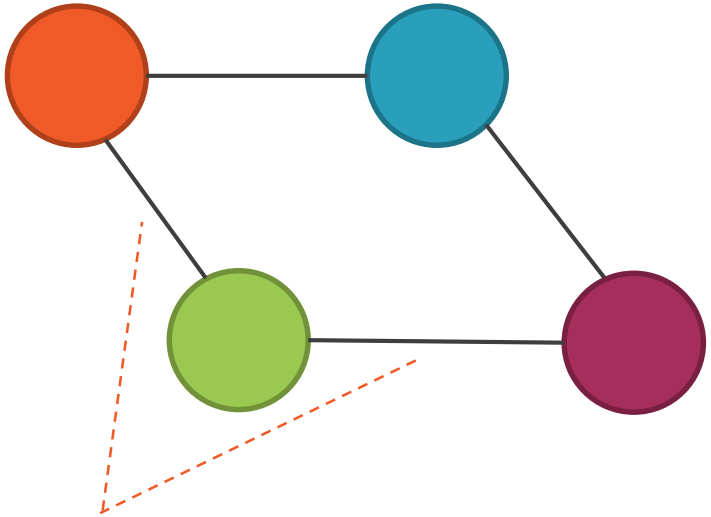
5 licenses + 5 discounts = 10 classes

The Bridge pattern replaces complexity multiplication with complexity addition.

Recap: Applying the Bridge Pattern

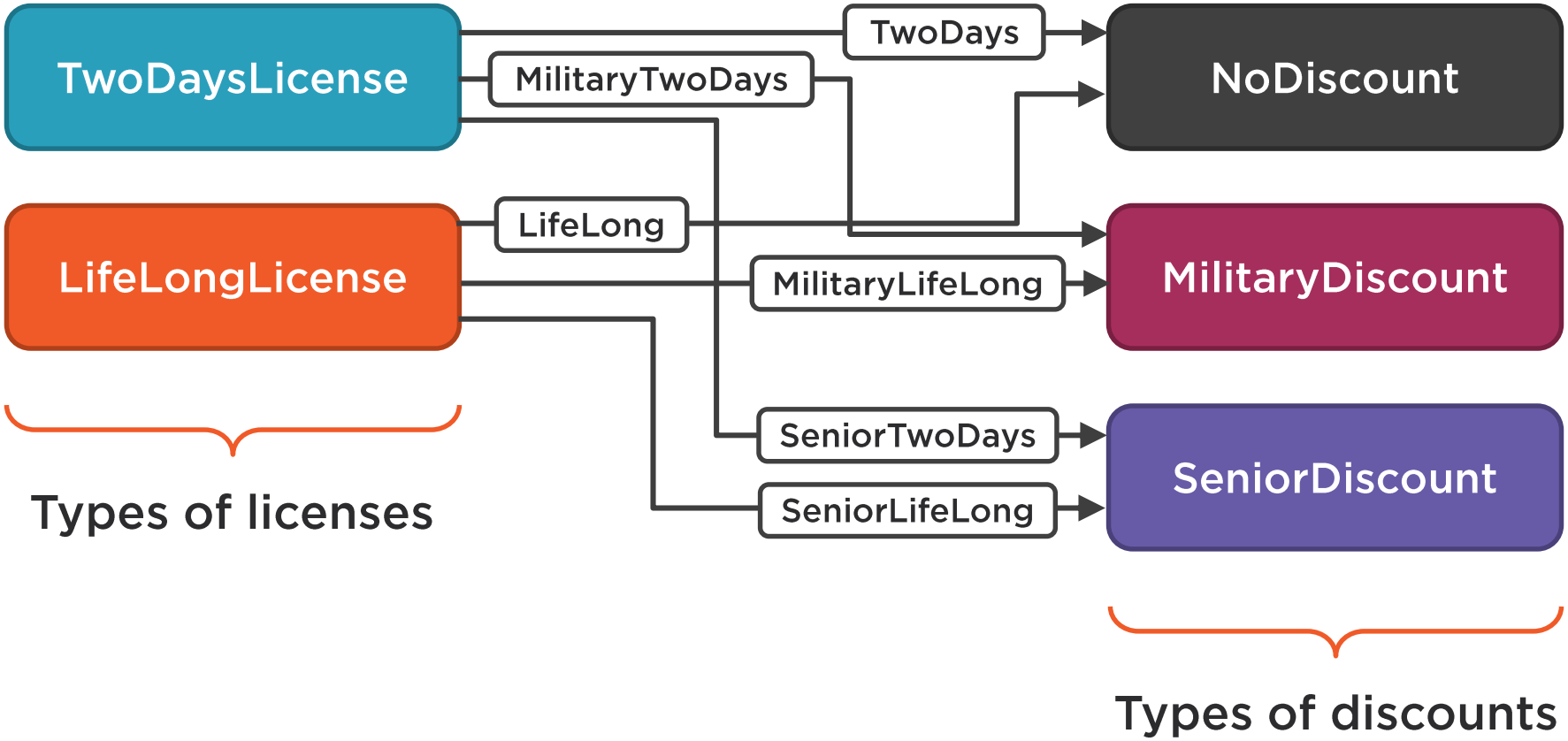


Complexity emerges from coupling



Coupling is connections between code elements

Recap: Applying the Bridge Pattern



Connection multiplication

Recap: Applying the Bridge Pattern

TwoDaysLicense

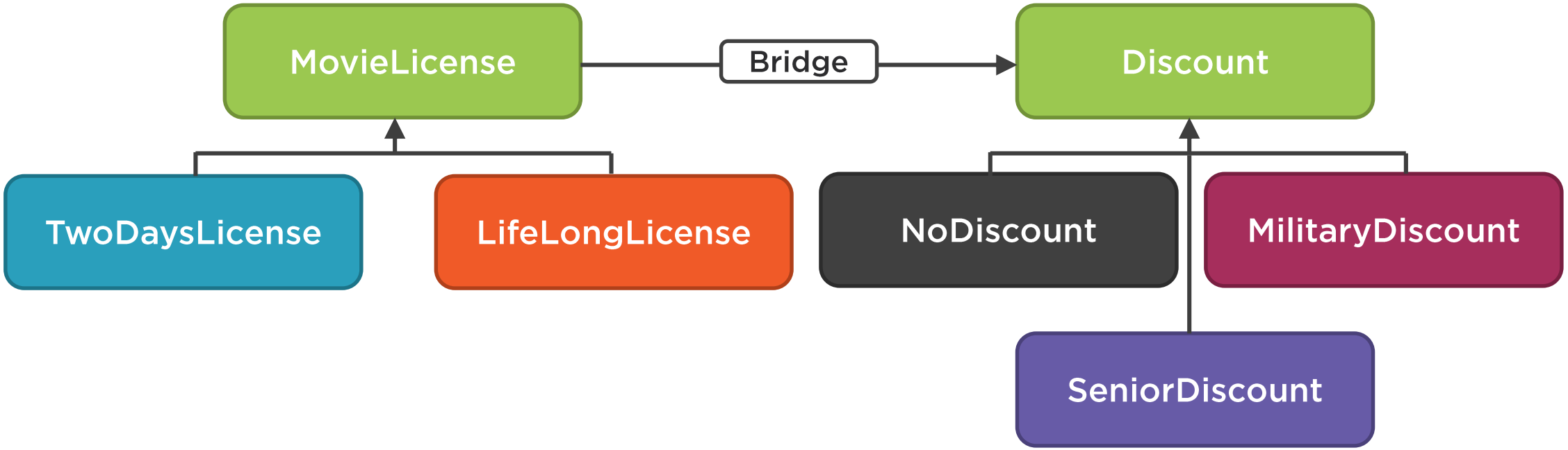
LifeLongLicense

NoDiscount

MilitaryDiscount

SeniorDiscount

Recap: Applying the Bridge Pattern

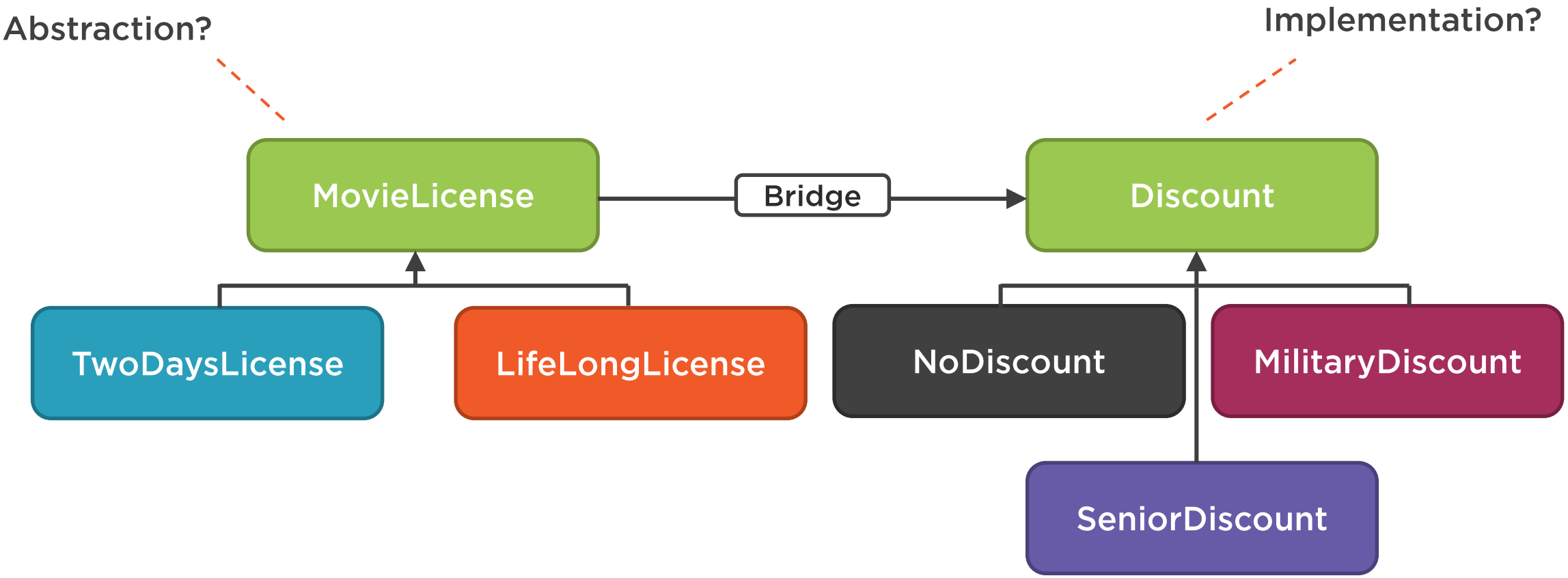


Connection addition

The Bridge Design Pattern

Its purpose is to decouple an abstraction from its implementation so that the two can vary independently.

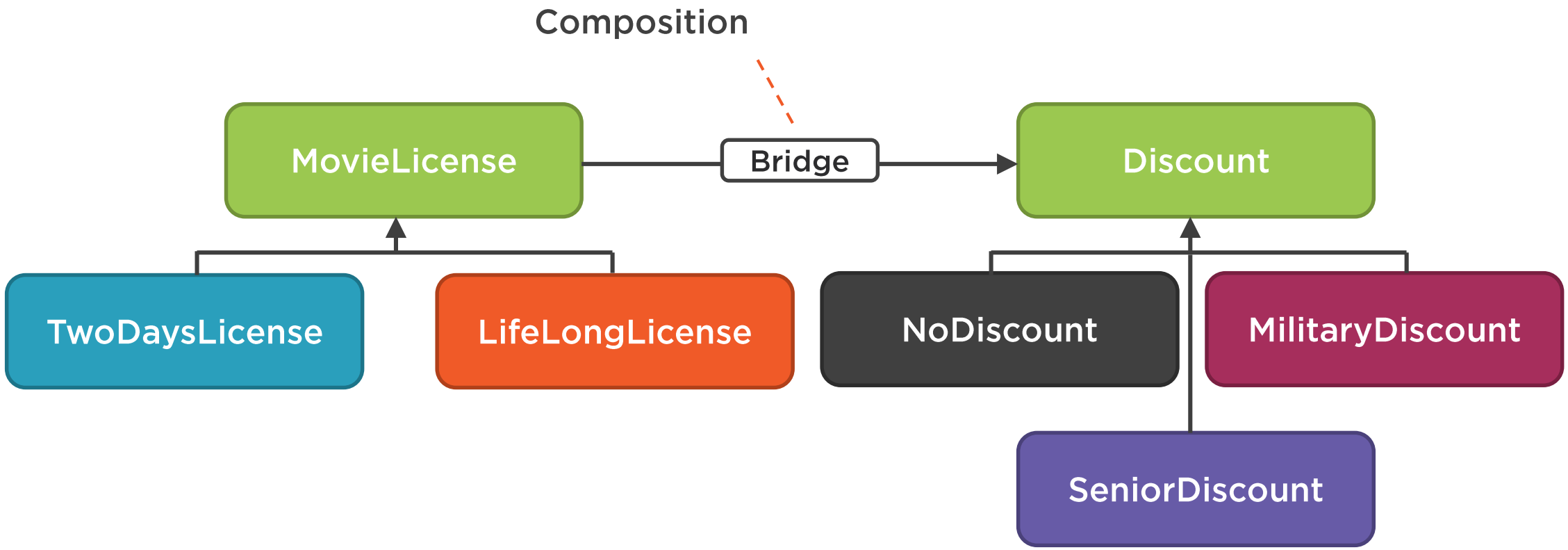
Recap: Applying the Bridge Pattern



The Bridge Design Pattern

Its purpose is to split a class hierarchy through composition to reduce coupling.

Recap: Applying the Bridge Pattern



The Alternative Implementation

```
public class Order
{
    public PaymentStatus PaymentStatus { get; }
    public DeliveryStatus DeliveryStatus { get; }
}
```

```
public enum PaymentStatus
{
    AwaitingPayment,
    Paid,
    PaymentFailed
}
```

```
public enum DeliveryStatus
{
    NotShipped,
    Shipped,
    Delivered
}
```

Different aspects of the order

The Alternative Implementation

```
public class Order
{
    public PaymentStatus PaymentStatus { get; }
    public DeliveryStatus DeliveryStatus { get; }
}

public enum PaymentStatus
{
    AwaitingPayment,
    Paid,
    PaymentFailed
}

public enum DeliveryStatus
{
    NotShipped,
    Shipped,
    Delivered
}
```

```
public enum Status
{
    AwaitingPaymentNotShipped,
    AwaitingPaymentShipped,
    AwaitingPaymentDelivered,
    PaidNotShipped,
    PaidShipped,
    PaidDelivered,
    PaymentFailedNotShipped,
    PaymentFailedShipped,
    PaymentFailedDelivered,
}
```

The number will also grow exponentially

The Alternative Implementation

Refactor toward composition

Recap: The Alternative Implementation

Bridge pattern

- Replaced inheritance with composition
- Moved all business logic to the base class
- Natural extension of the Bridge pattern

Prefer composition over inheritance.

Recap: The Alternative Implementation

Composition is more flexible than inheritance

- Easier to change the behavior
- Inheritance is rigid

Composition is easier to understand

Recap: The New Requirement

Inject a new enumeration into the class

```
public enum LicenceType
{
    TwoDays,
    LifeLong
}
```

```
public enum Discount
{
    None,
    Military,
    Senior
}
```

```
public enum SpecialOffer
{
    None,
    TwoDaysExtension
}
```

Loosely coupled dimensions

Course Summary

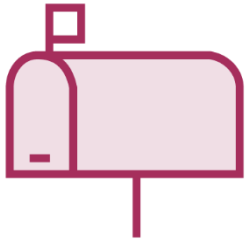
"Decouple an abstraction from its implementation so that the two can vary independently"

Split a class hierarchy through composition to reduce coupling

- Helps prevent combinatorial explosion

Refactor inheritance toward composition

Contacts



http://bit.ly/vlad-updates

@vkhorikov

https://enterprisecraftsmanship.com