Behavioral Pattern: Observer

Kevin Dockx
Architect

@KevinDockx https://www.kevindockx.com

Coming Up

Describing the observer pattern

- Service communication in a ticket management system

Structure of the observer pattern

Coming Up

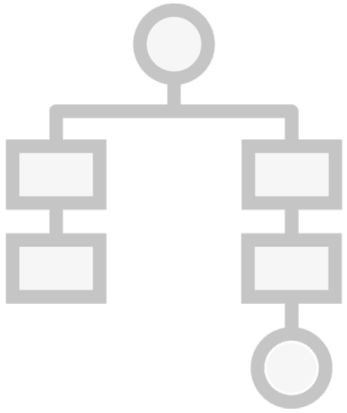
Use cases for this pattern

Pattern consequences

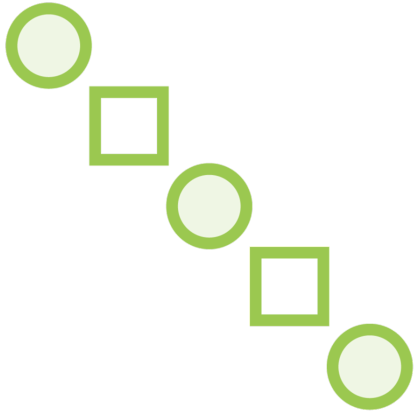
Related patterns

Describing the Observer Pattern

Creational



Structural



Behavioral

Observer

The intent of this pattern is to define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Describing the Observer Pattern

Observer is a common pattern

- Observables in Angular
- Service communication in microservice architectures
- ...

```
public class OrderService
{ }

public class TicketStockService
{ }

public class TicketResellerService
{ }
```

Describing the Observer Pattern

```
public class OrderService
{ }

public class TicketStockService
{ }

public class TicketResellerService
{ }
```

Describing the Observer Pattern

```
public class OrderService
{ }

public class TicketStockService
{ }

public class TicketResellerService
{ }
```

Describing the Observer Pattern

```
public class OrderService
{ }

public class TicketStockService
{ }

public class TicketResellerService
{ }
```

Describing the Observer Pattern

These services are related and need to maintain consistency

```
public class OrderService
{
     private TicketStockService _ticketStockService;
     private TicketResellerService _ticketResellerService;
}

public class TicketStockService
{ }

public class TicketResellerService
{ }
```

Describing the Observer Pattern

```
public class OrderService
{
     private TicketStockService _ticketStockService;
     private TicketResellerService _ticketResellerService;

     // methods to notify services...
}

public class TicketStockService
{ }

public class TicketResellerService
{ }
```

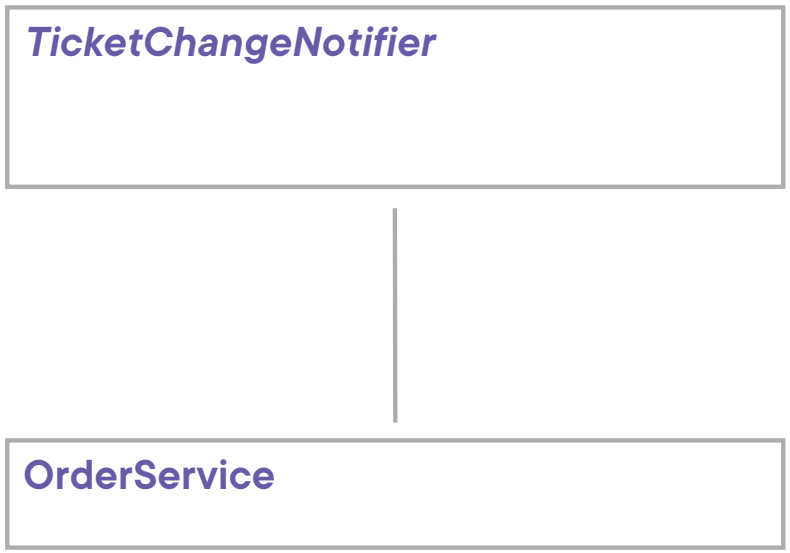
Describing the Observer Pattern

We're introducing tight coupling
Becomes complex to maintain

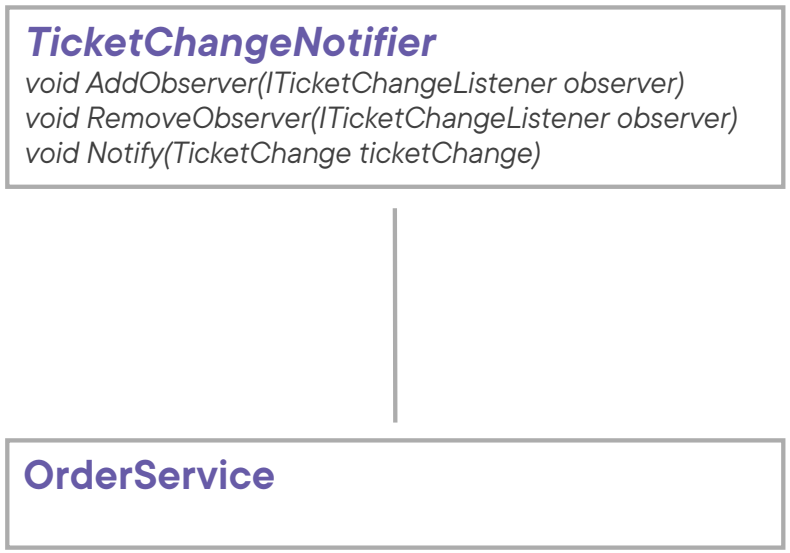
Describing the Observer Pattern

Describing the Observer Pattern

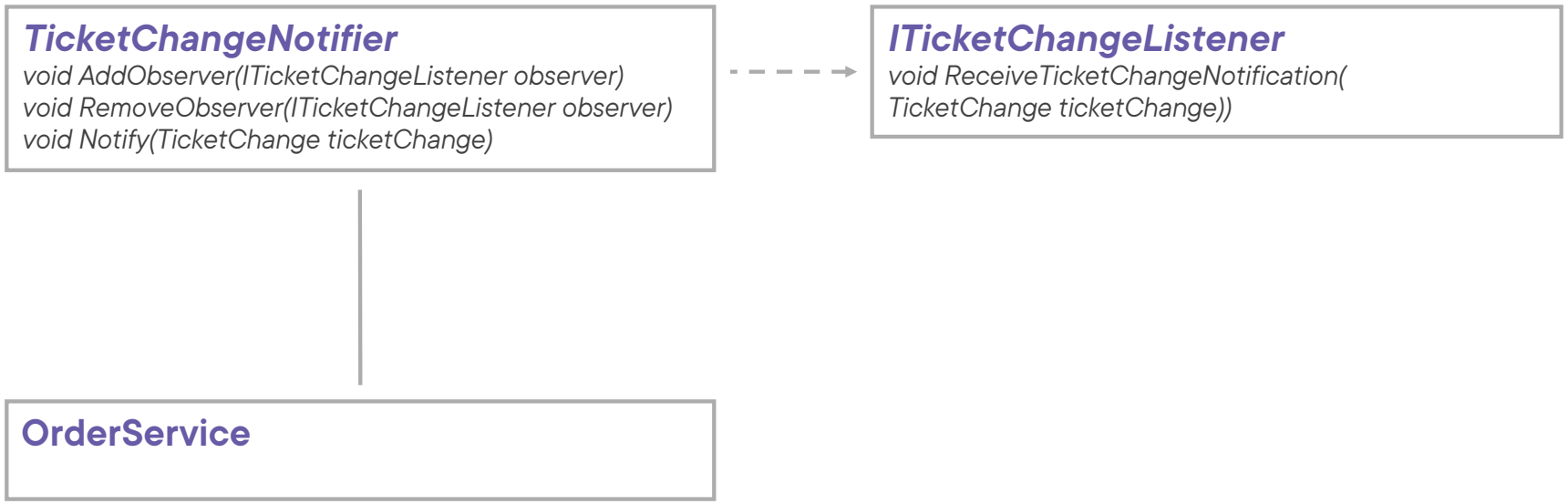
Describing the Observer Pattern



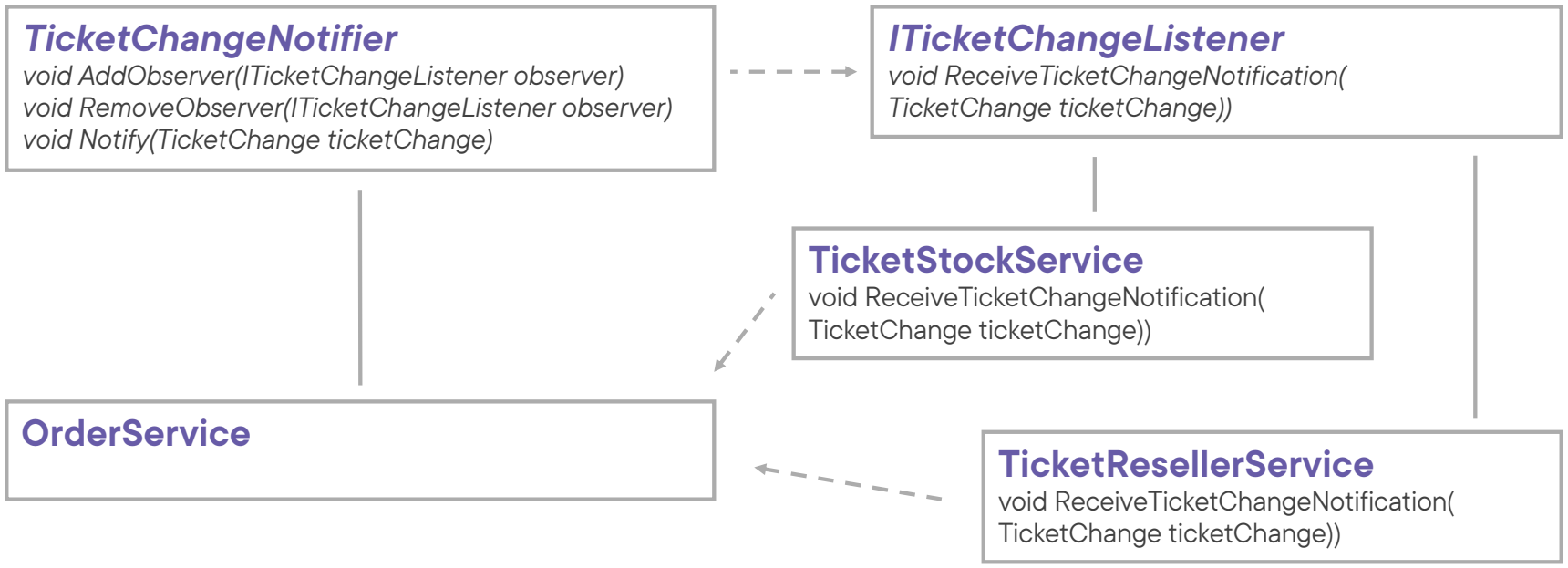
Describing the Observer Pattern



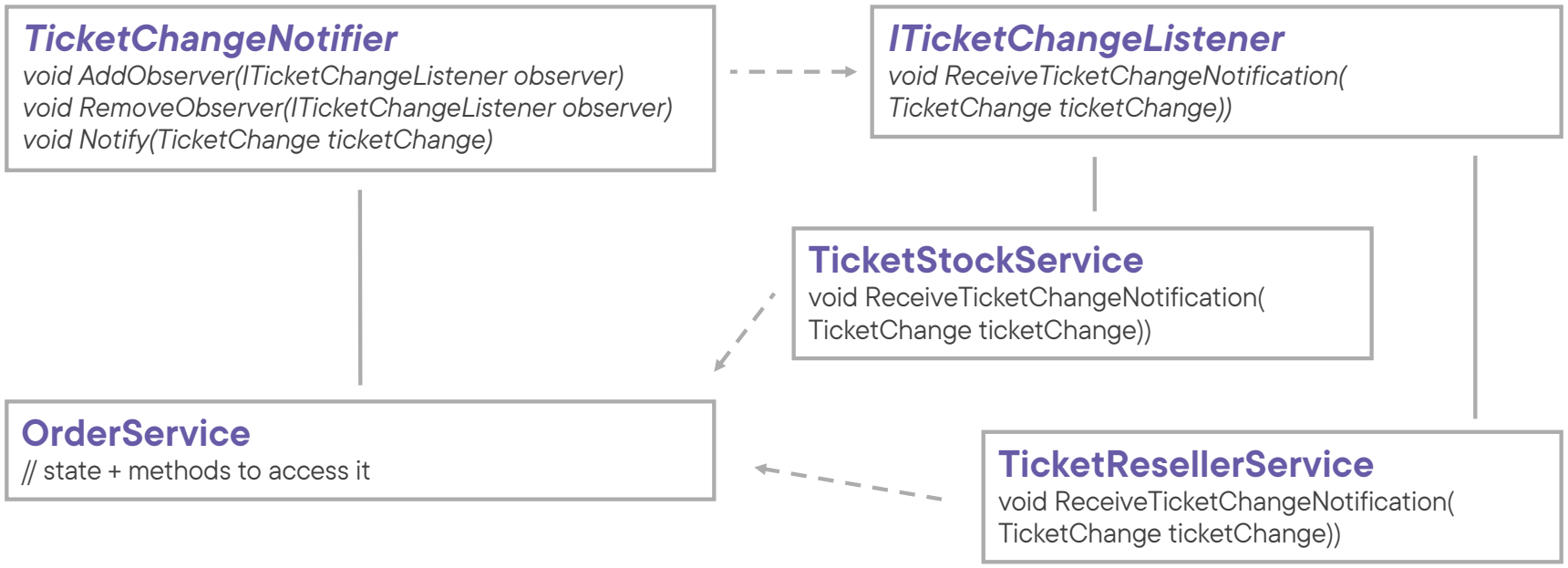
Describing the Observer Pattern



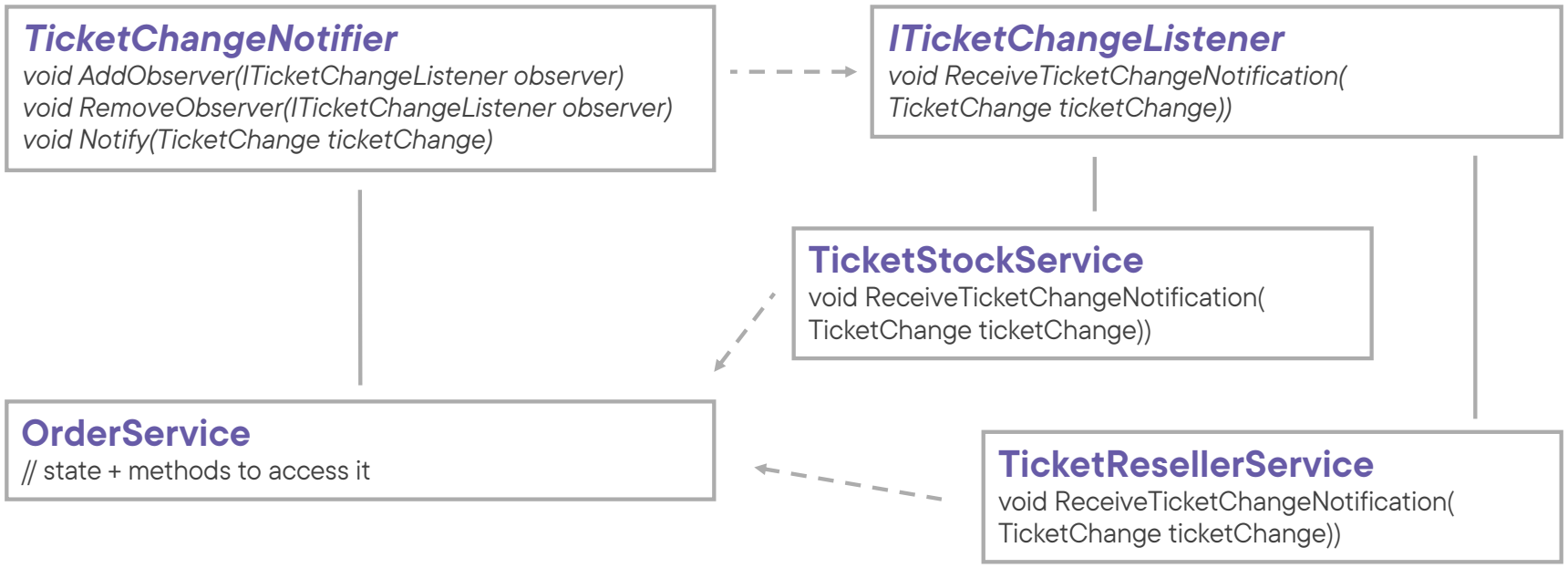
Describing the Observer Pattern



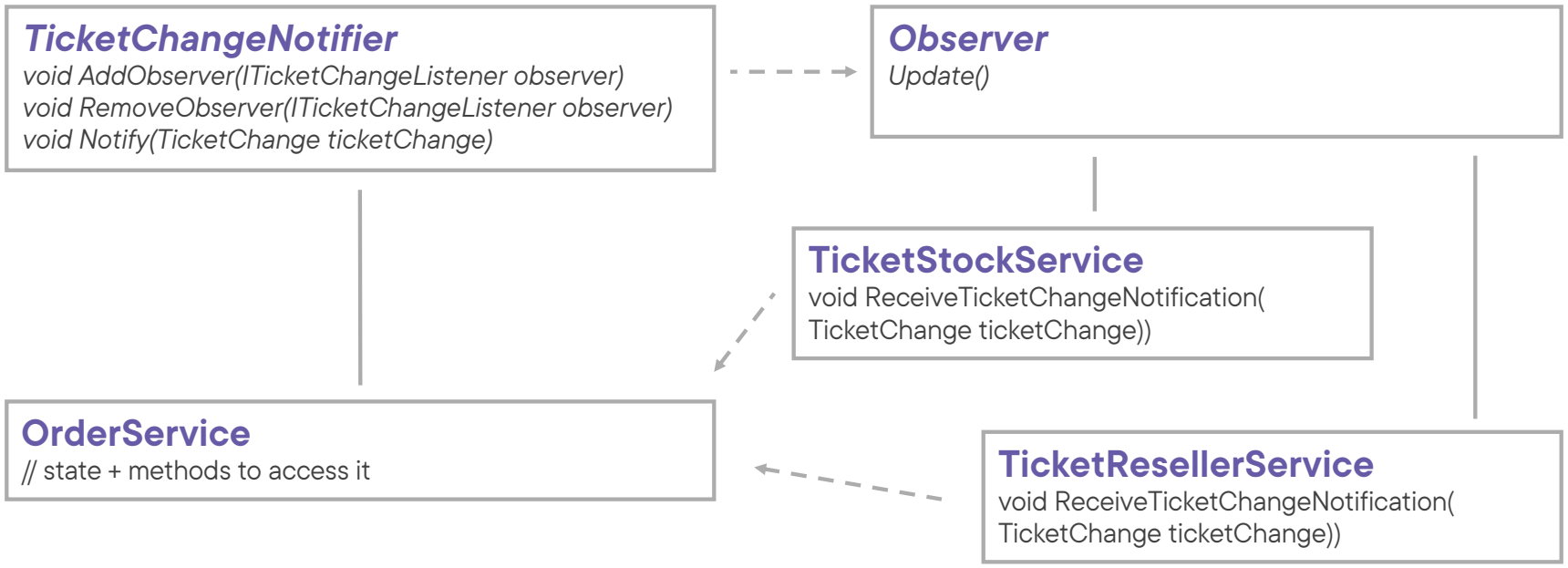
Describing the Observer Pattern



Structure of the Observer Pattern

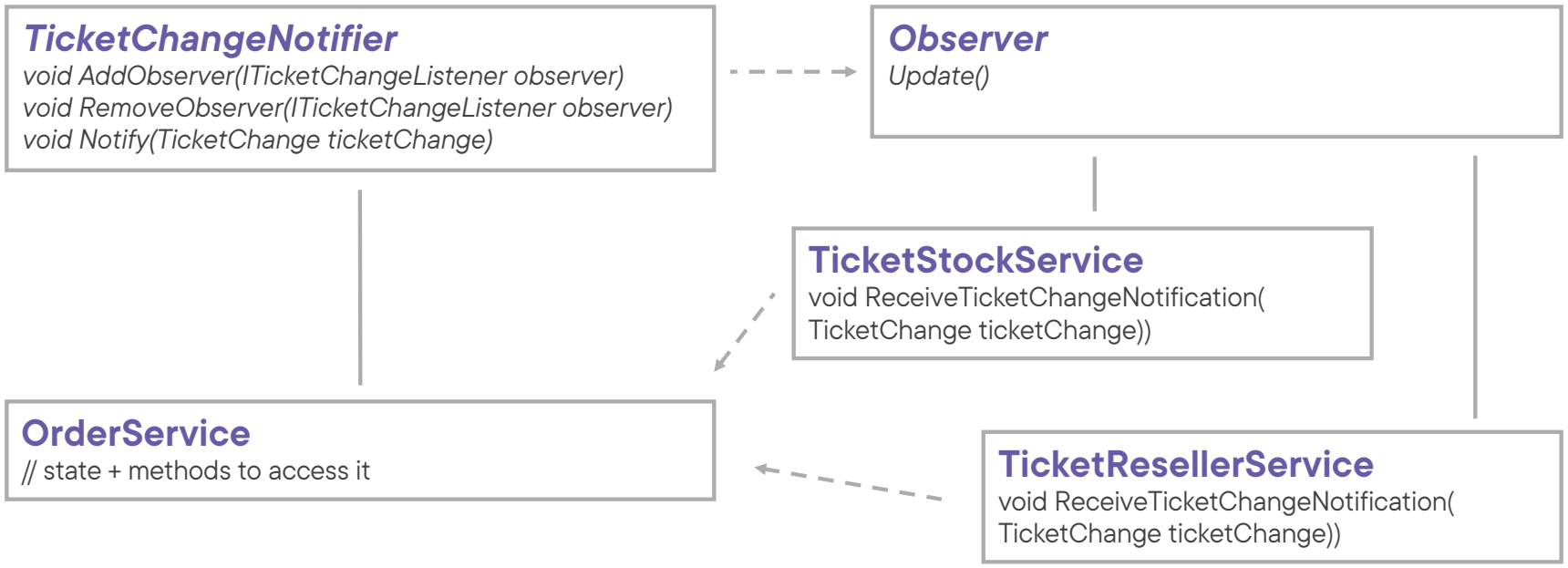


Structure of the Observer Pattern

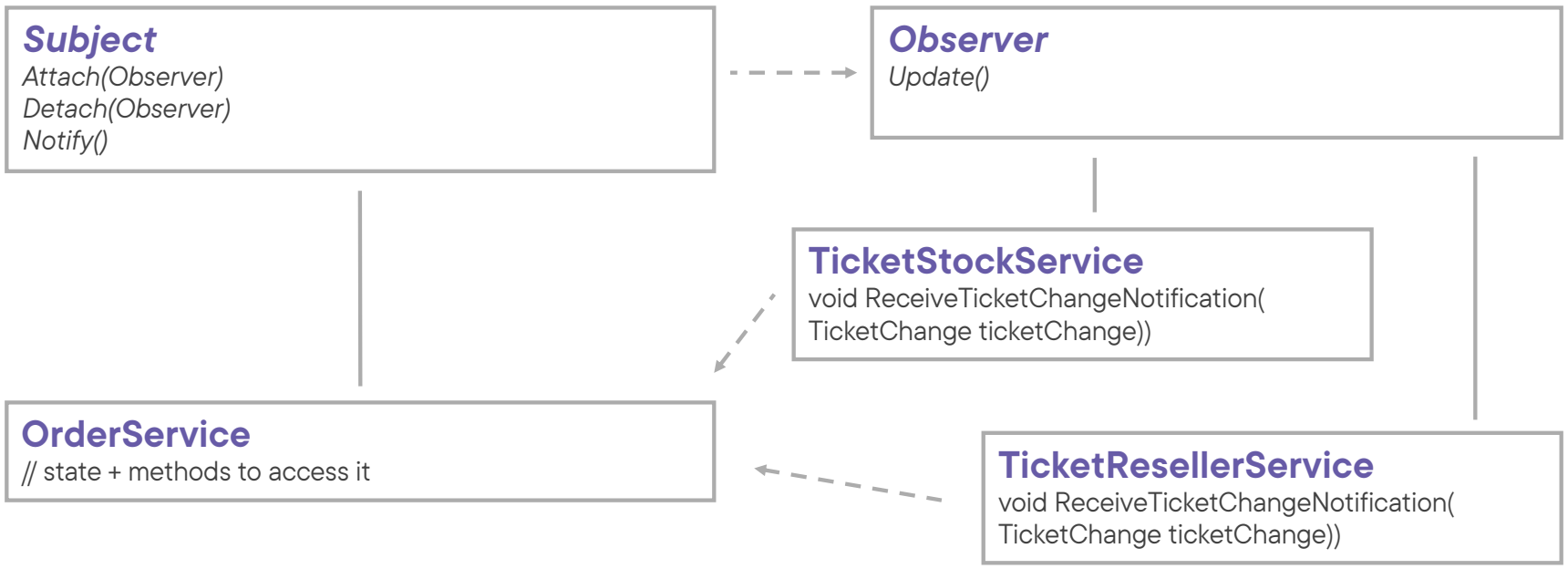


Observer defines an updating interface for objects that should be notified of changes in a **Subject**

Structure of the Observer Pattern

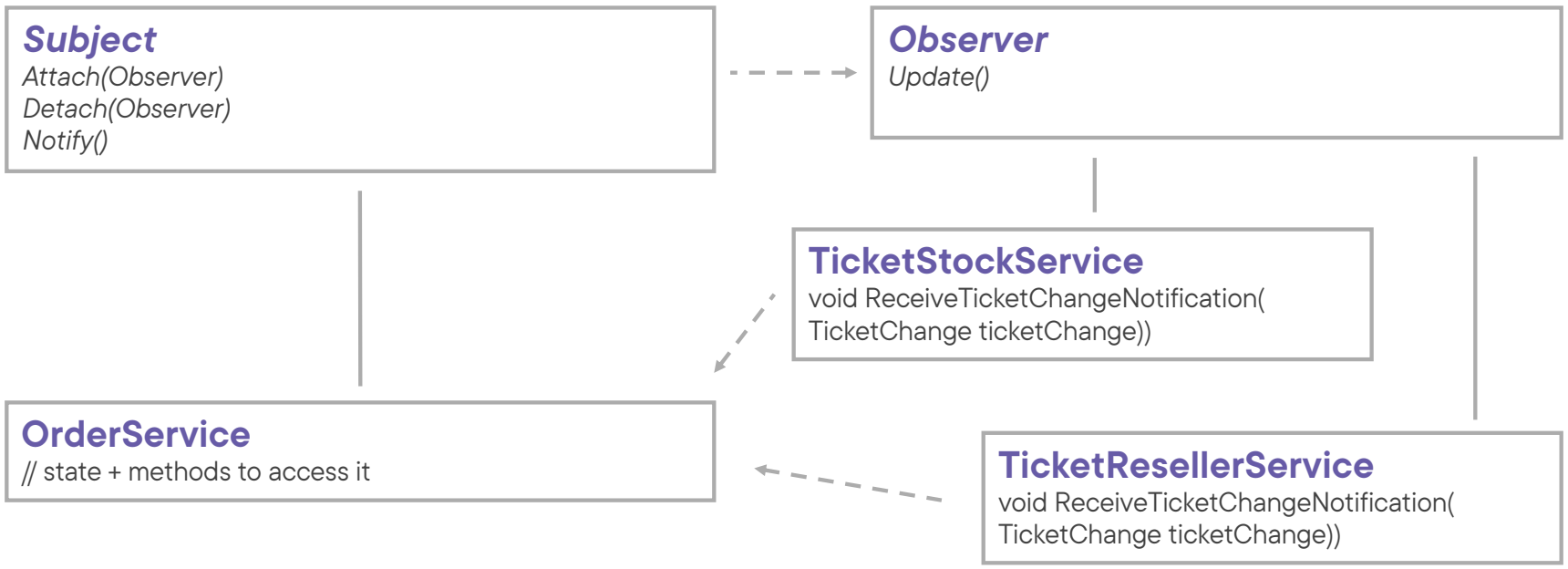


Structure of the Observer Pattern

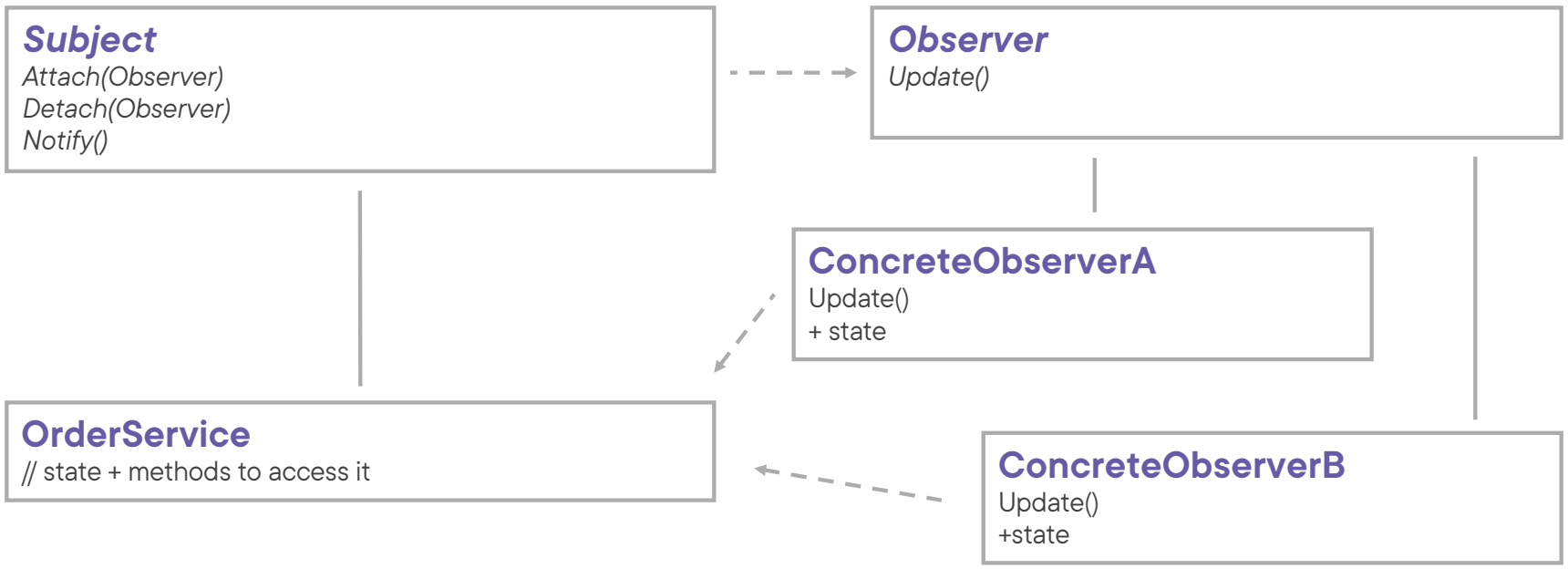


Subject knows its **Observers**.
Provides an interface for attaching and detaching them.

Structure of the Observer Pattern



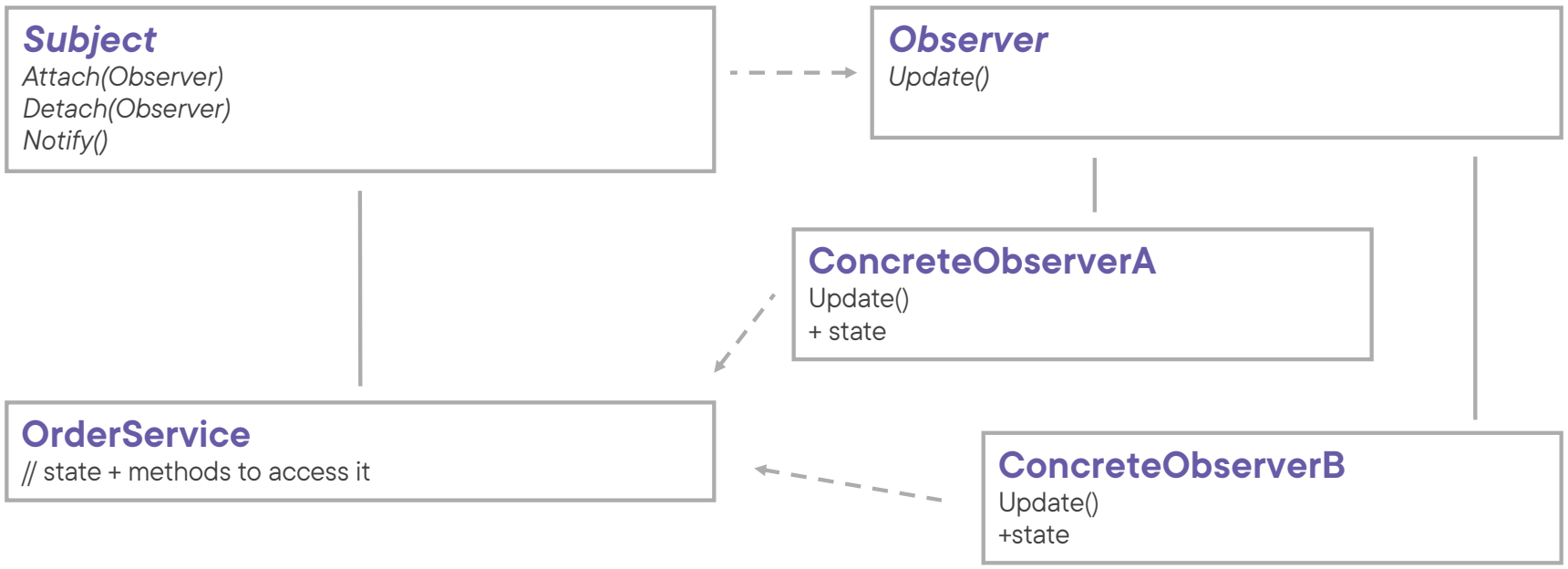
Structure of the Observer Pattern



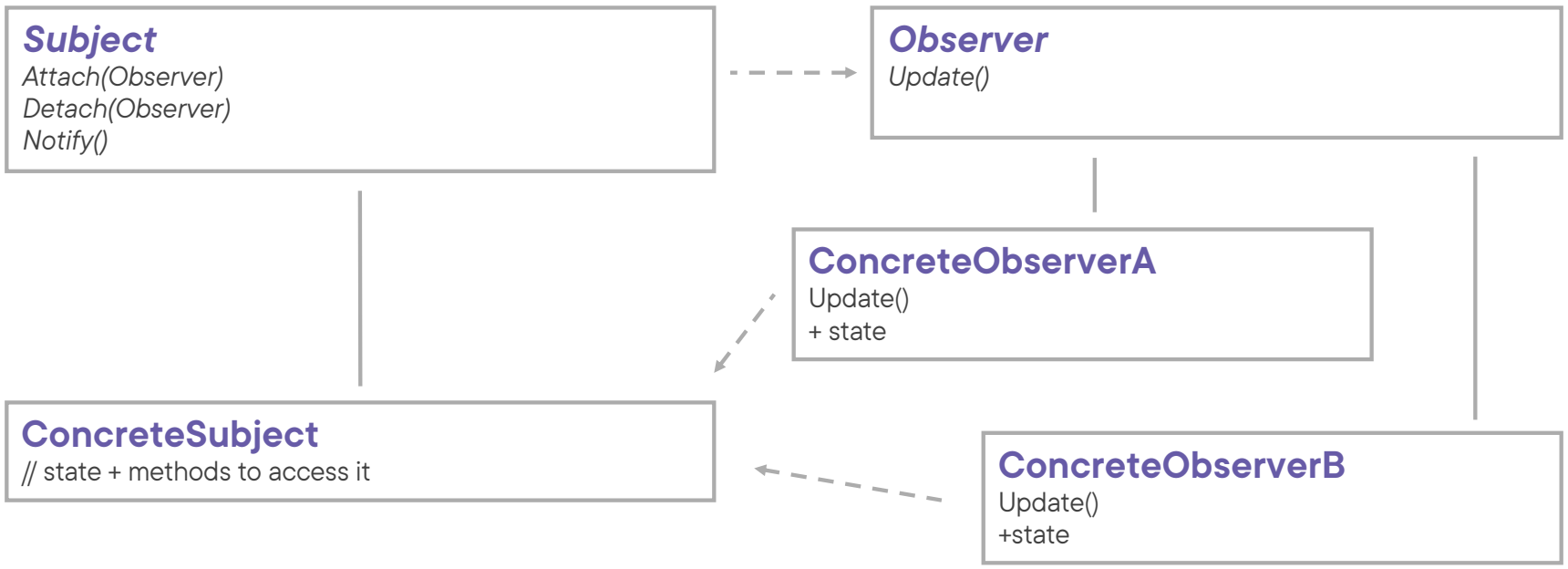
ConcreteObserver store state that must remain consists with the **Subjects**' state. They implement the **Observer** updating interface to keep state consistent.

ConcreteSubject stores state of interest to **ConcreteObserver** objects, and sends a notification to its **Observers** when its state changes

Structure of the Observer Pattern



Structure of the Observer Pattern



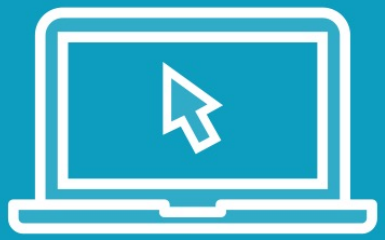
Structure of the Observer Pattern

State is passed through via the Notify method

- No need for the ConcreteObserver to hold a reference to the ConcreteSubject

Both implementations are valid

Demo



Implementing the observer pattern

Use Cases for the Observer Pattern

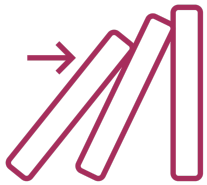
When a change to one object requires changing others, and you don't know in advance how many objects need to be changed

When objects that observe others are not necessarily doing that for the total amount of time the application runs

When an object should be able to notify other objects without making assumptions about who those objects are

Pattern Consequences

It allows subjects and observers to vary independently: subclasses can be added and change without having to change others: open/closed principle

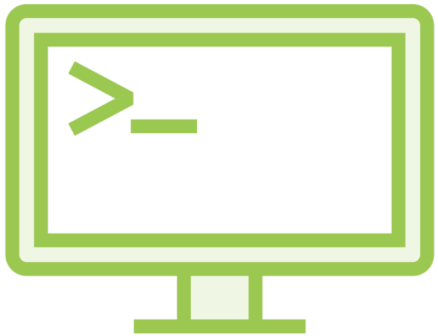


Subject and observer are loosely coupled: open/closed principle

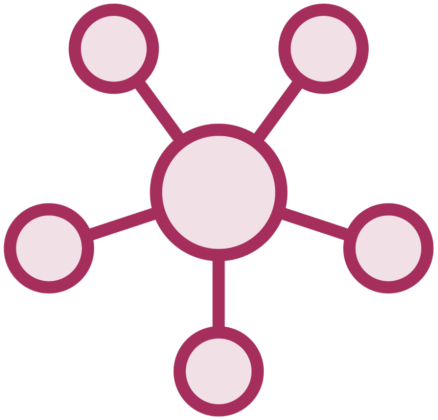
It can lead to a cascade of unexpected updates

Related Patterns

Chain of Responsibility
Passes a request along a chain of receivers



Command
Connects senders with receivers unidirectionally



Mediator
Eliminates direct connections altogether

Observer
Allows receivers of requests to (un)subscribe at runtime

Summary

Intent of the observer pattern:

- To define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Summary

Implementation:

- Use an abstract base class to implement `Notify`, `AddObserver` and `RemoveObserver` functionality
- ConcreteSubjects are responsible for managing their state

Up Next:
Behavioral Pattern: State