Software design patterns

GoF Observer

"The critical design tool for software development is a mind well educated in design principles"

version: 1.0.5

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

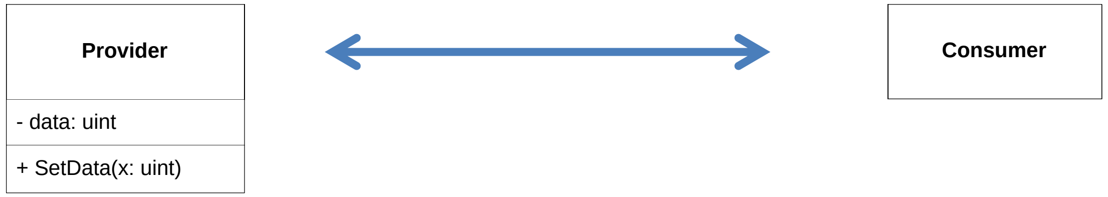
GoF Observer

Decoupling the general “data-update” problem

`Provider` contains `data` which is changed every now and them by some entity (other than the Consumer)

How can we realize this coupling?

`Consumer` uses `data` and would like to do some update based on changes in `data`



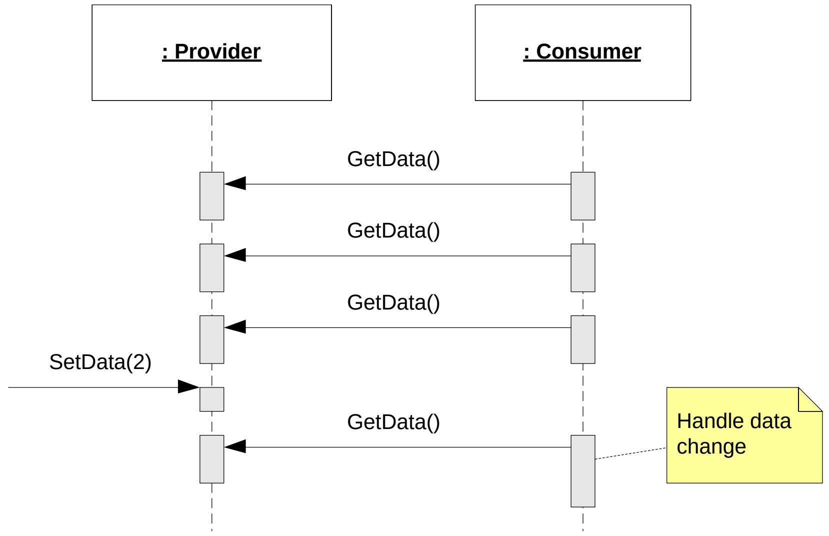
AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Decoupling the general "data-update" problem

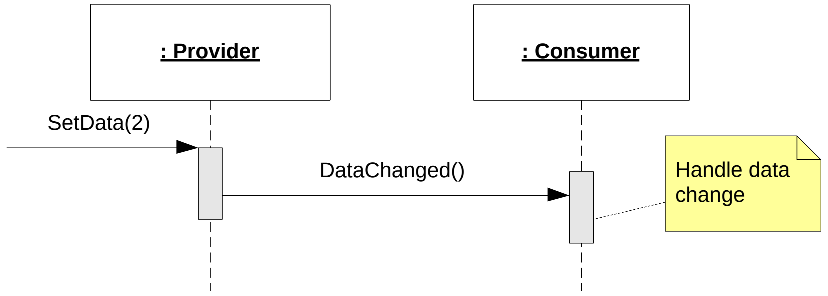
Consumer polls Provider

[Example]

Consider: What happens if more consumers are added?



Provider notifies Consumer when data changes



AARHUS UNIVERSITY SCHOOL OF ENGINEERING

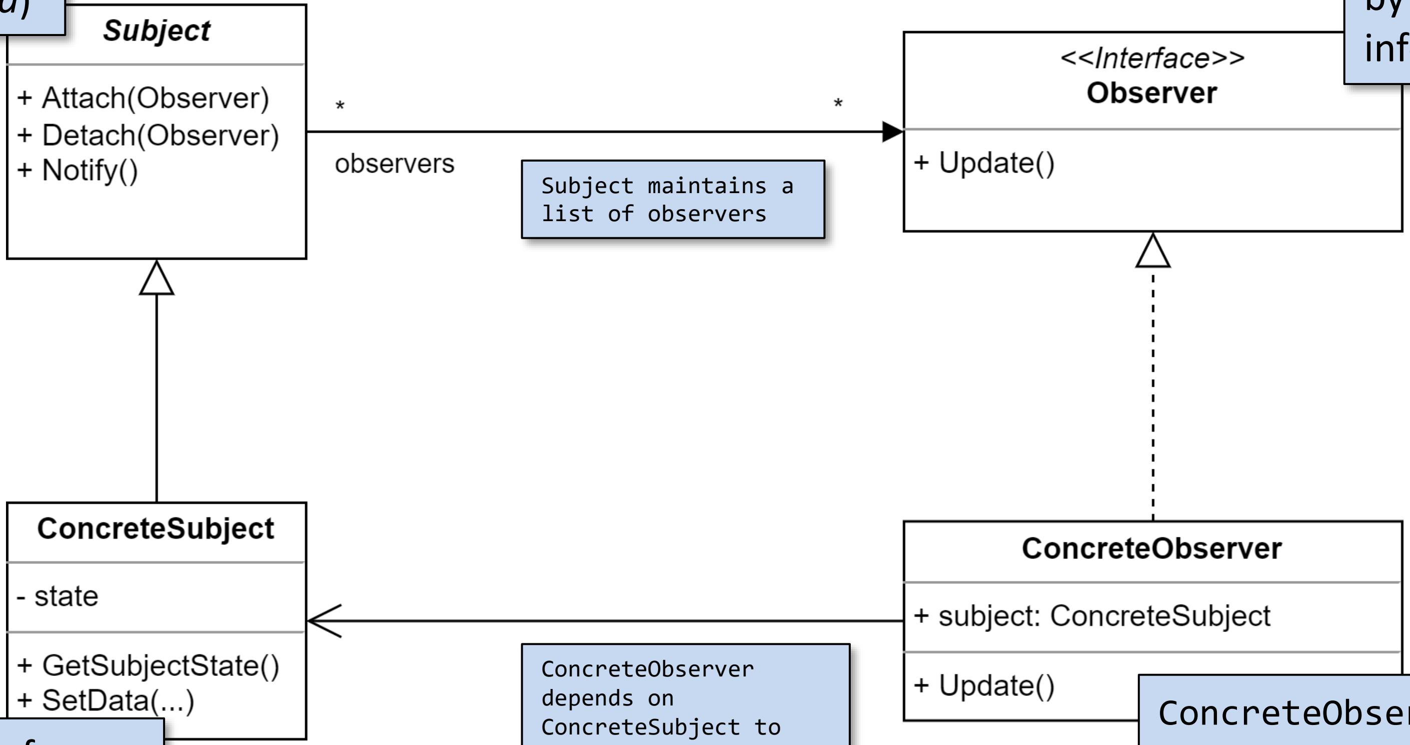
GoF Observer to the rescue!

- We need some mechanism that…
  - allows `Consumers` to be added to the `Provider` without changing the `Provider` (i.e. adhere to OCP)
  - allows `Provider` to inform `Consumers` of data changes (i.e. promotes low coupling)
  - allows many `Consumers` to be informed on updates of same data

- We need….GoF Observer!

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

GoF Observer: Structure



`Subject` is an **abstract base** class for all data subjects (i.e. things that get *updated*)

`Observer` is an **interface** that must be implemented by all classes that wish to be informed of data changes

`Subject maintains a list of observers`

`ConcreteObserver depends on ConcreteSubject to GetState`

`ConcreteSubject` inherits from `Subject`. This is the actual class that must be monitored (in our case, `Provider`)

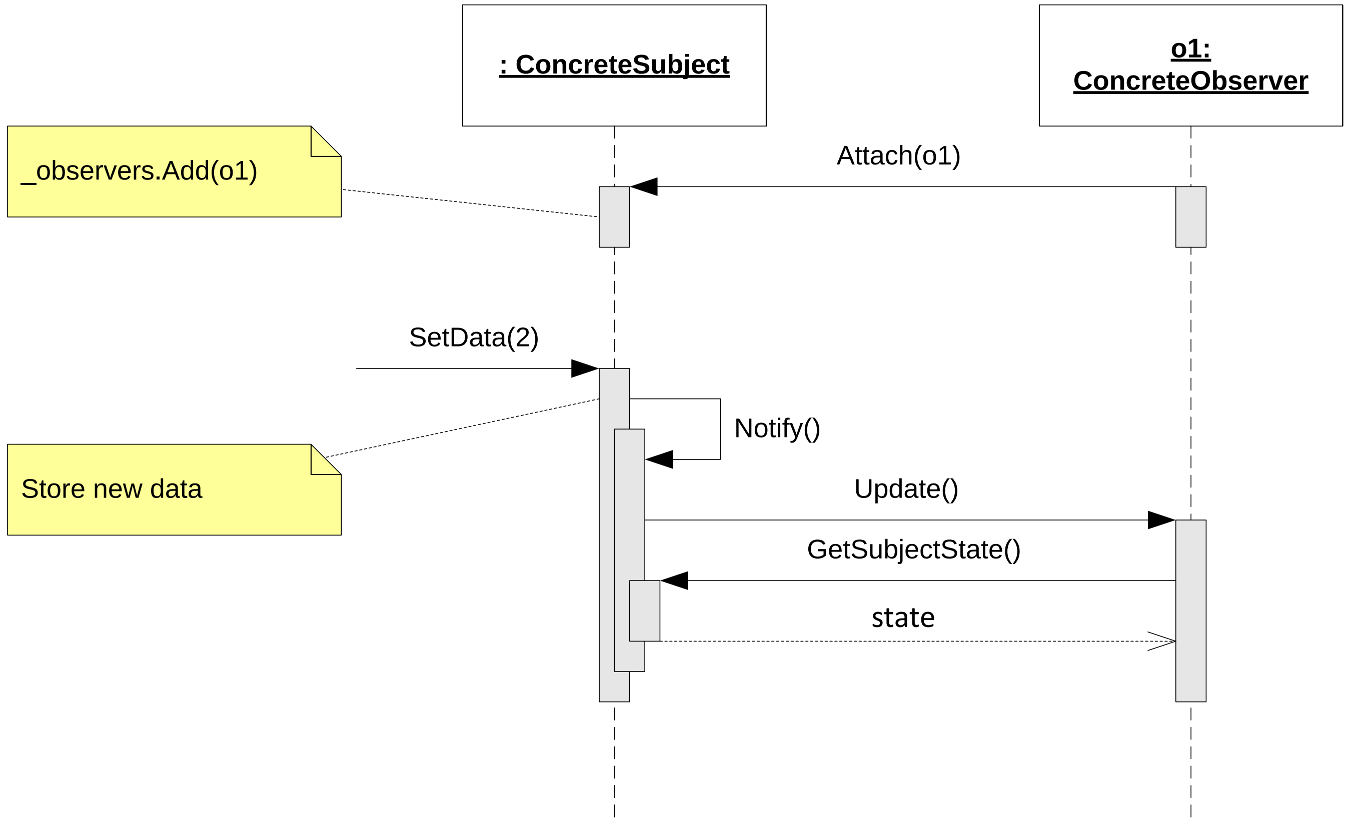
`ConcreteObserver` implements the `Observer` interface. This is the actual class that must receive updates (in our case, `Consumer`)

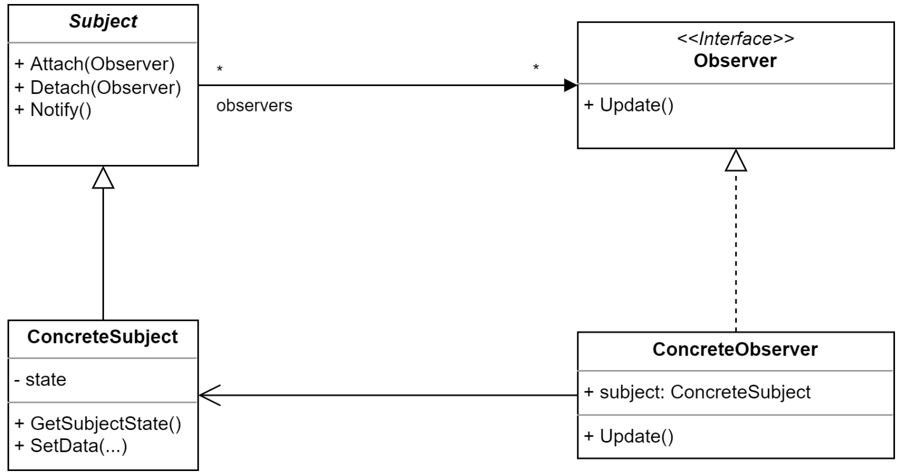
Diagram Source

SCHOOL OF ENGINEERING

GoF Observer: Behavior – Pull Variant

- *Pull* variant
  (Observer *pulls* state from subject)



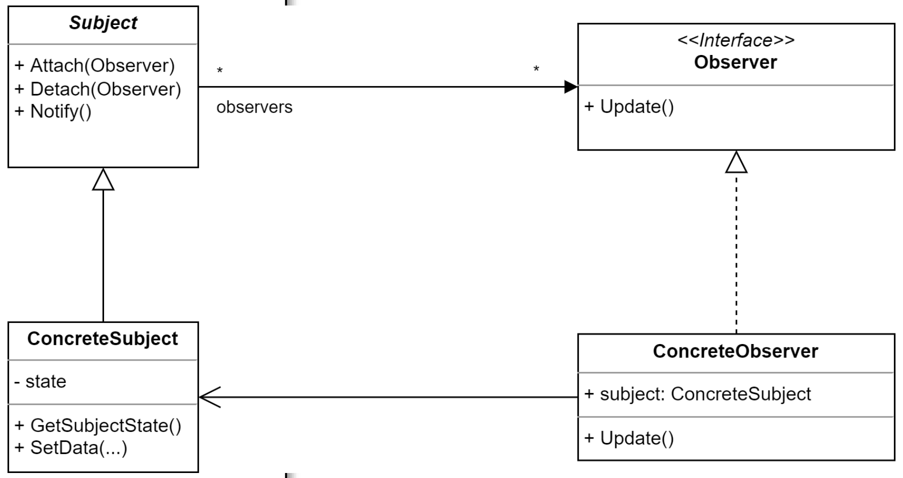


AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Observer pattern in C# - Pull Variant

Interfaces

```
 public class SubjectData
 {
    public int Measurement { get; set; }
 }

 public interface ISubject
 {
     void Attach(IObserver obs);
     void Detach(IObserver obs);
     void Notify();
}

 public interface IObserver
 {
    void Update();
 }
```

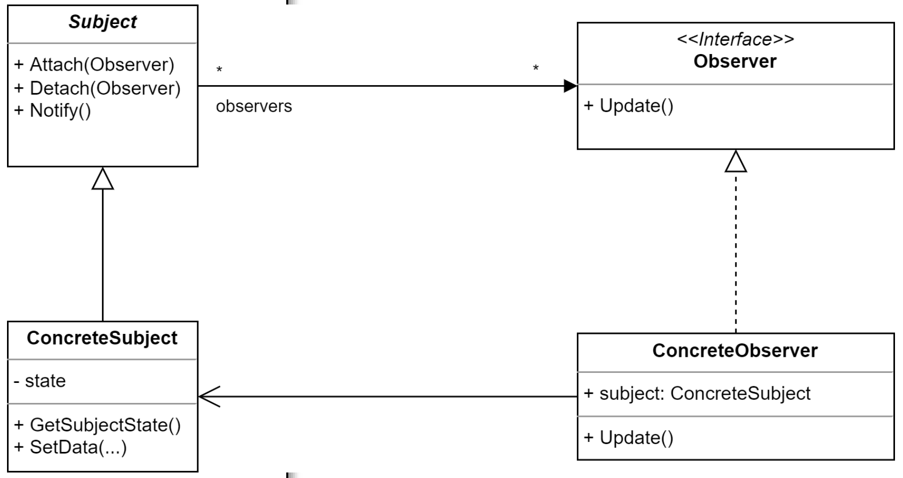


AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Observer pattern in C# - Pull Variant

Subject implementation

```
public class ConcreteSubject : ISubject
{
   private List<IObserver> observers = new List<IObserver>();
   private SubjectData state = new SubjectData();

   public void Attach(IObserver obs) { observers.Add(obs); }
   public void Notify()
   {
       foreach (var observer in observers)
       {
           observer.Update();
       }
   }

   public SubjectData GetSubjectState()
   {
       return state;
   }
```

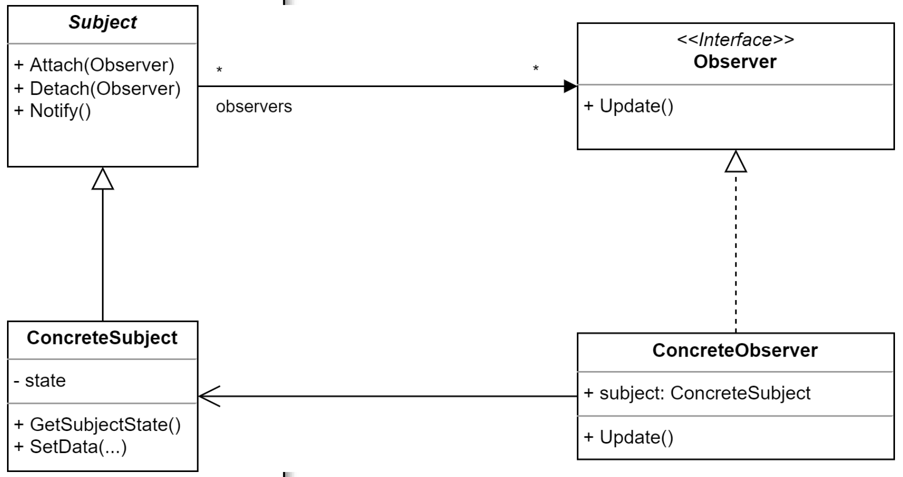


AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Observer pattern in C# - Pull Variant

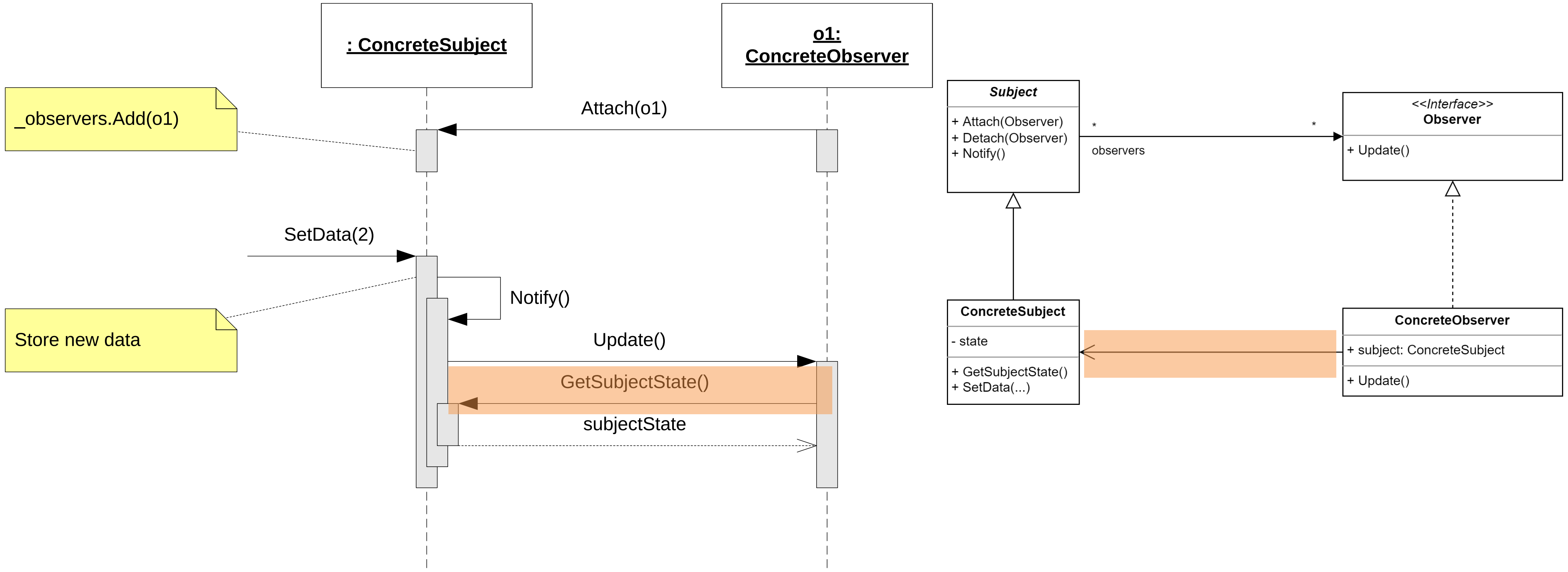
Observer implementation

```
public class ConcreteObserver : IObserver
{
   private ConcreteSubject subject;

   public ConcreteObserver(ConcreteSubject subject) {…}

   public void Update()
   {
       // Get subject data
       SubjectData newData = this.subject.GetSubjectState();
       // Handle new value of subject data
   }
```



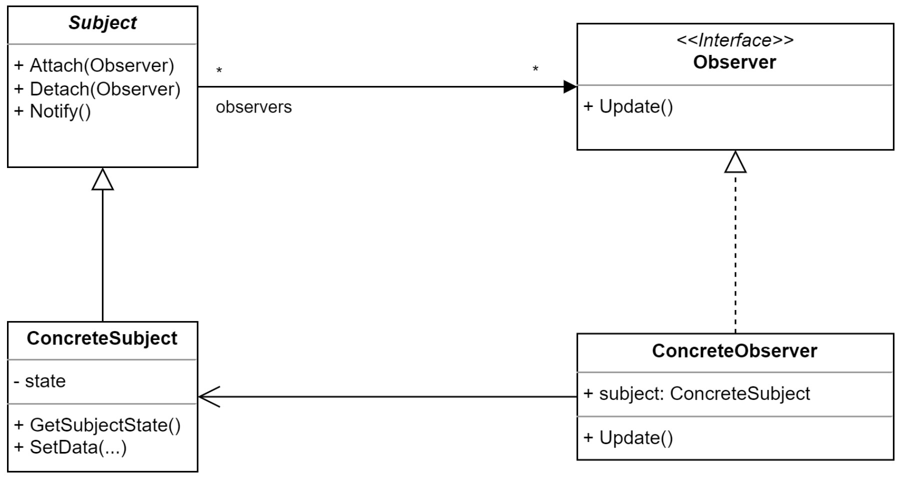
AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

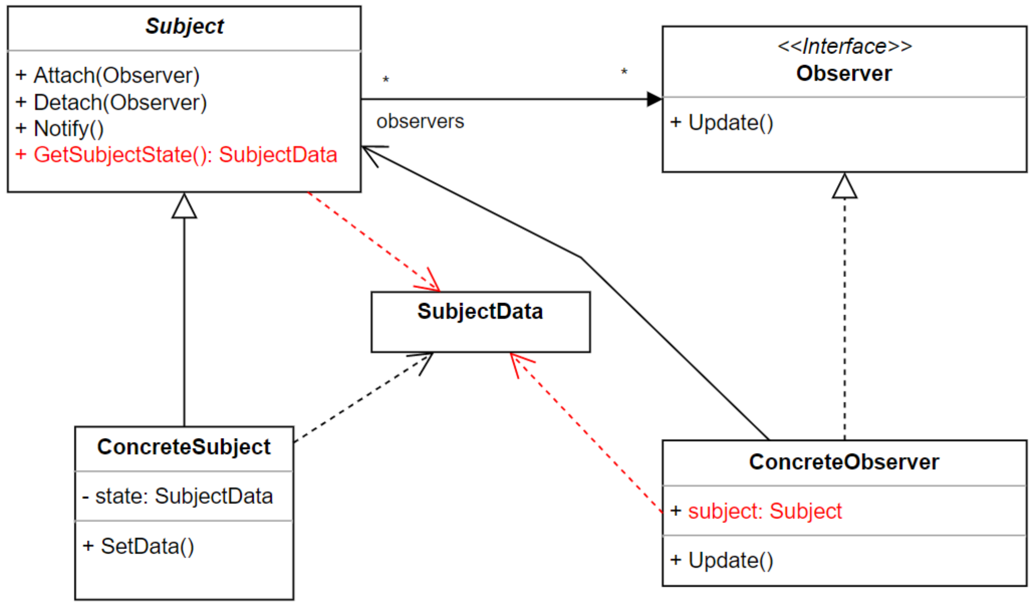
GoF Observer: Behavior – Pull Variant Problems



AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Discussion: Attempt at Decoupling ConcreteSubject and ConcreteObserver



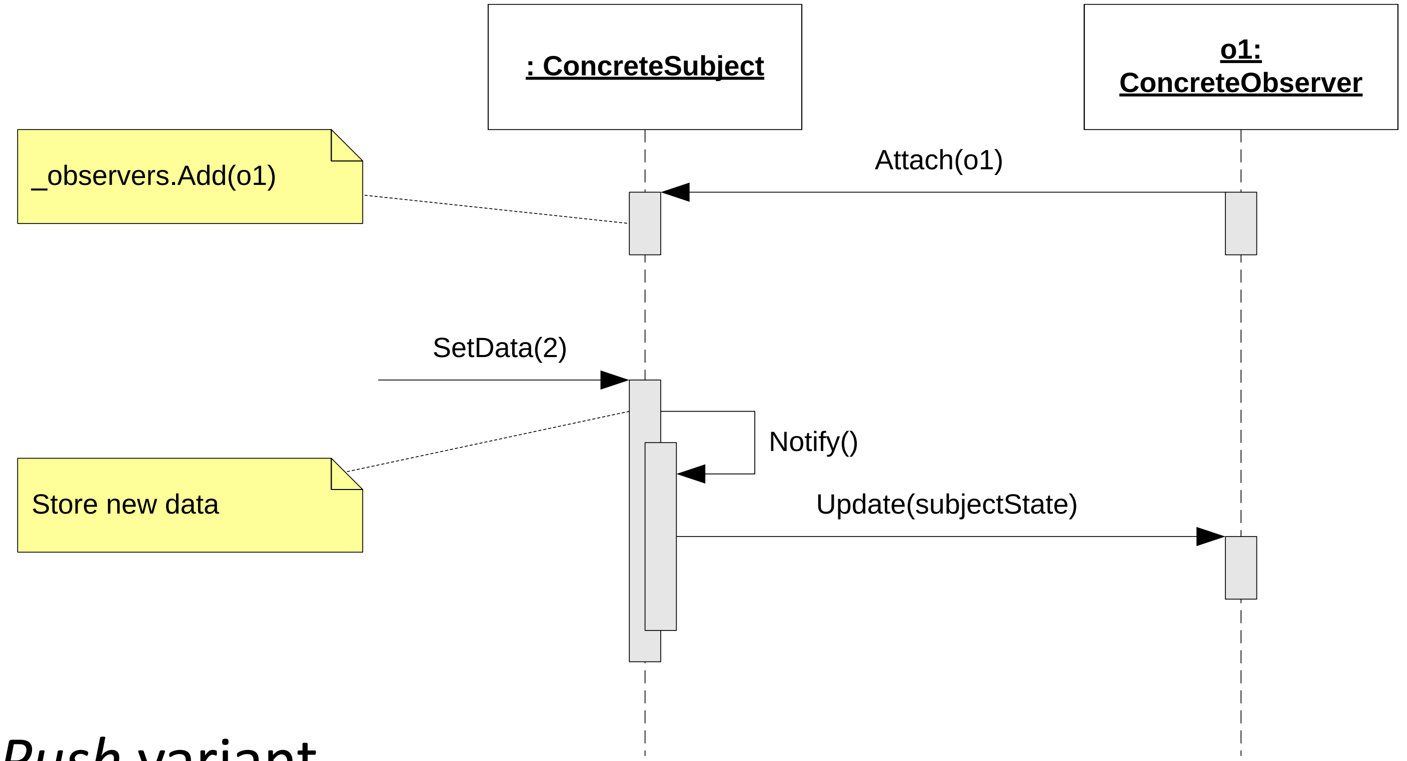


[Diagram Source]()

Is this a good design? Hint: does it apply the SR principle?

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

GoF Observer: Behavior – Push Variant



Question: What is the class diagram of this code?

- *Push* variant
  (Subject *pushes* state to observer)

GoF Observer: Behavior – Push Variant

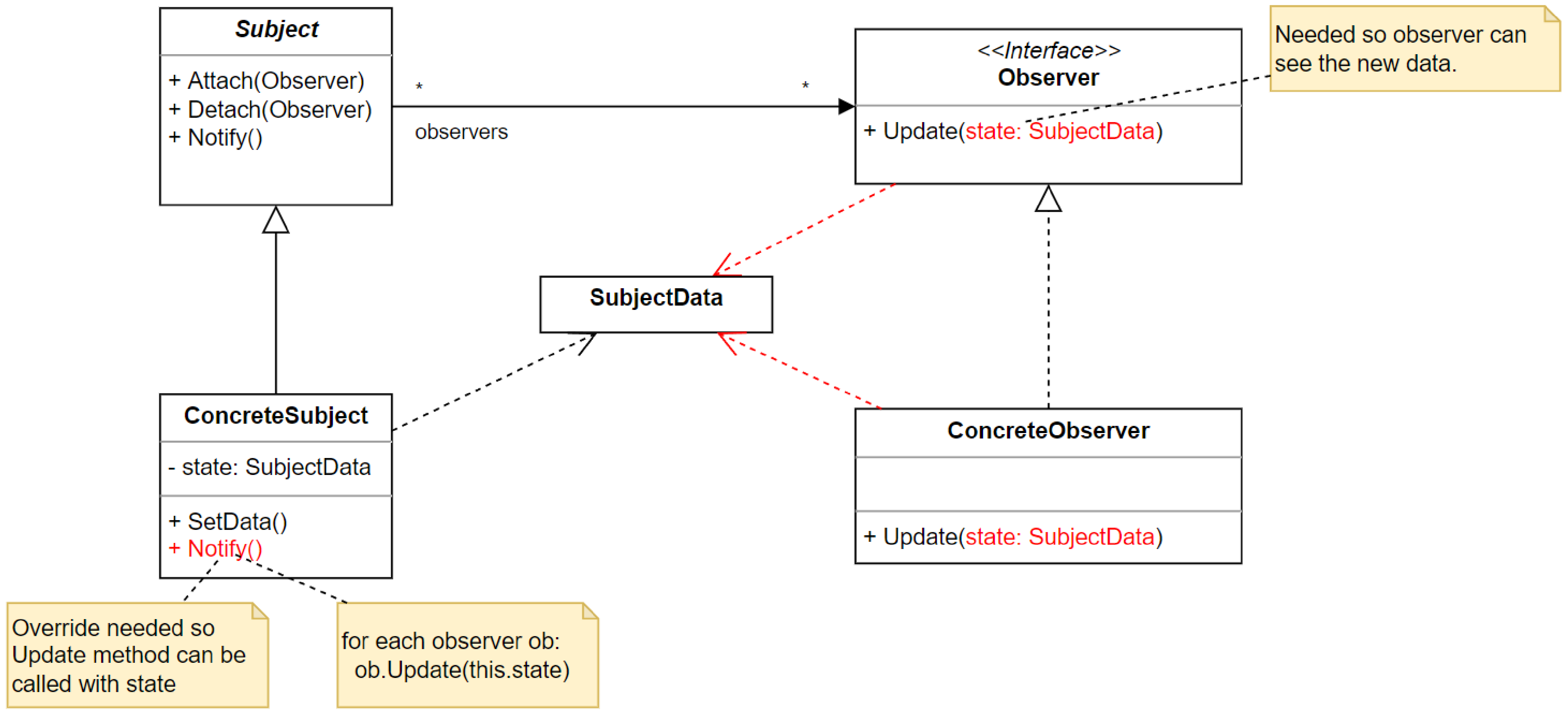


Diagram Source

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Observer pattern in C# - push

Interfaces

```
public class SubjectData
{
   public int Measurement { get; set; }
}

public interface ISubject
{
   void Attach(IObserver obs);
   void Detach(IObserver obs);
   void Notify();
}

public interface IObserver
{
   void Update(SubjectData subjectData);
}
```

Observer pattern in C# - push

Subject implementation

```
public class ConcreteSubject : ISubject
{
   private List<IObserver> observers = new List<IObserver>();
   private SubjectData state = new SubjectData();

   public void Attach(IObserver obs)
   {
       observers.Add(obs);
   }

   public void Notify()
   {
       foreach (var observer in observers)
       {
           observer.Update(state);
       }
   }
```

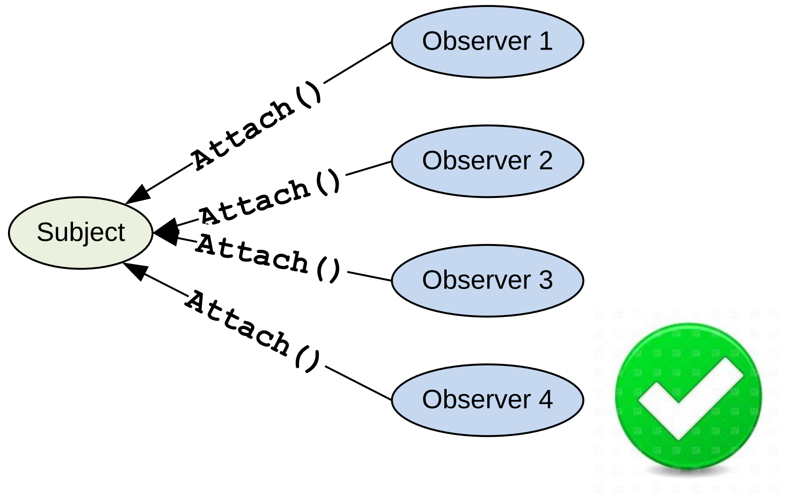
Observer pattern in C# - push

Observer implementation

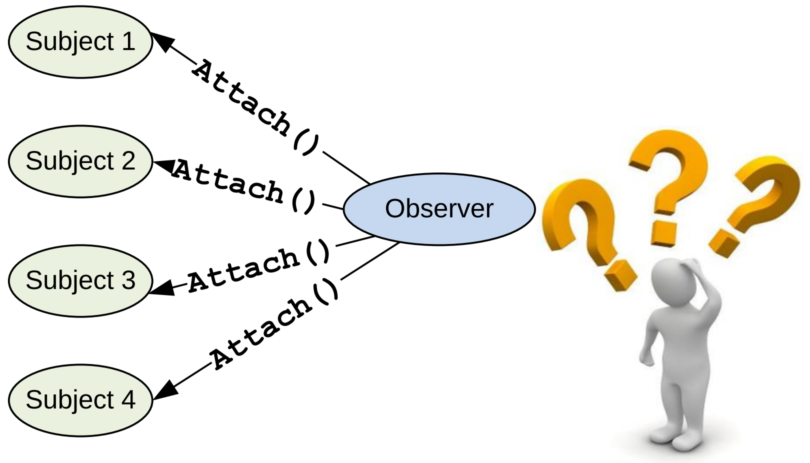
```
public class ConcreteObserver : IObserver
{
   public ConcreteObserver(ISubject subject)
   {
       subject.Attach(this);
   }

   public void Update(SubjectData subjectData)
   {
       // Handle new value of subject data
       ...
   }
```

GoF Observer – handling several subjects of same type

- The variant of GoF Observer we have studied handles several observers registering on the *same* subject



- How about one Observer attaching to several subjects of the **same** type?



AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Handling several subjects of same type – Subject sends reference-to-self

```
class SomeSubject : Subject
{
  public void SetState(State state)
  {
    _state = state;
    NotifyObservers(this);
  }
}
```

```
class SomeObserver : Observer
{
  public void AddSubject(Subject s)
  {
    s.Attach(this);
  }

  public void Update(Subject s)
  {
    // Do something with ‘s’
  }

}
```

Sending `this` uniquely ID’s the `Subject`

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Handling several subjects of same type – Subject sends tag

```
class SomeSubject : Subject
{
  string tag;

  public void SetState(State state)
  {
    _state = state;
    NotifyObservers(tag);
  }
}
```

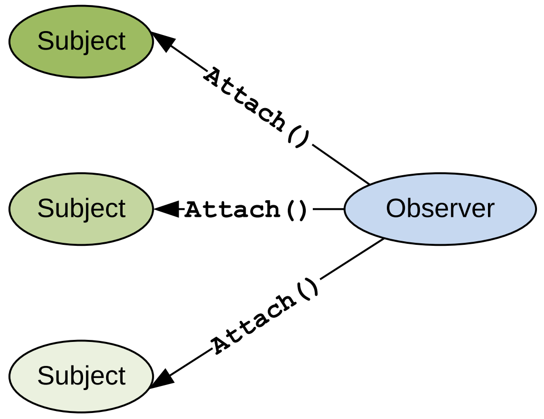
```
class SomeObserver : Observer
{
  public void AddSubject(Subject s)
  {
    s.Attach(this);
  }

  public void Update(string tag)
  {
    // Do something with the Subject
    // ID'ed by 'tag'
  }

}
```

Sending `tag` also ID's the `Subject`, but does not send an object reference. It is up to the `Observer` to find the correct reference

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

GoF Observer – handling subjects of different types

- How can we handle observers that connect to subjects of *different* types?
  - Again: Notice `Subject` is a generic **abstract base class**
  - `IObserver` as a generic **interface**!



AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

GoF Observer – handling subjects of different types “manually”

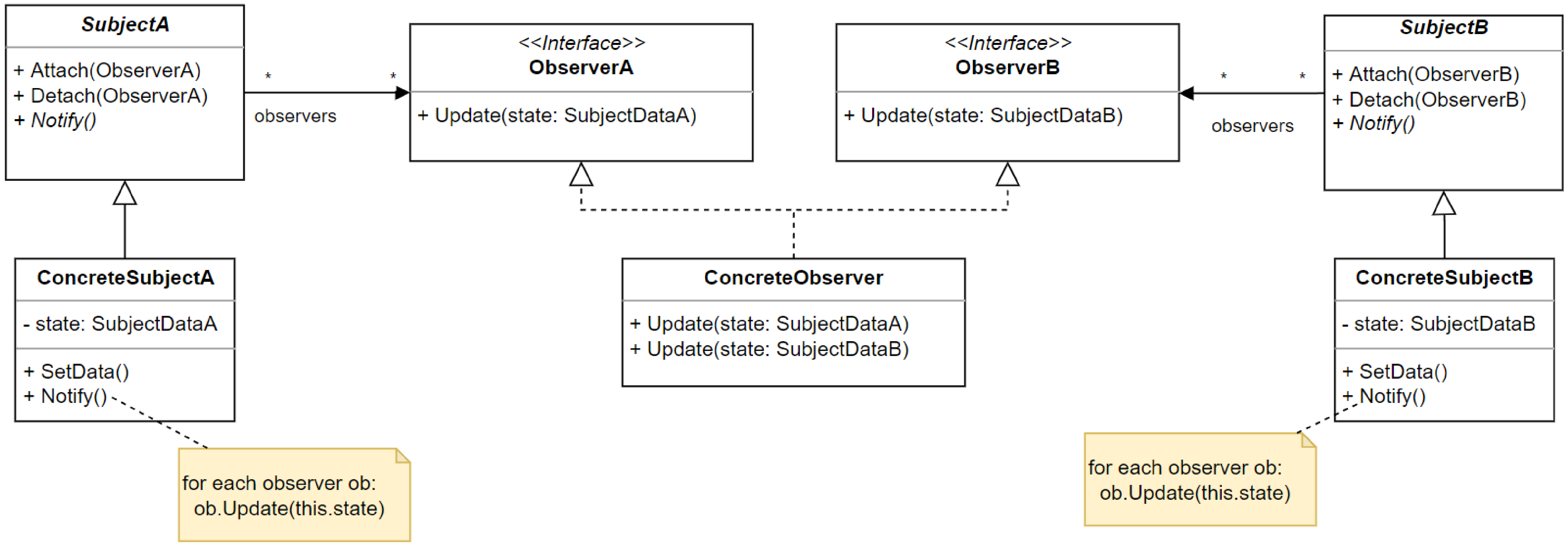


Diagram Source

GoF Observer – handling subjects of different types with generic types

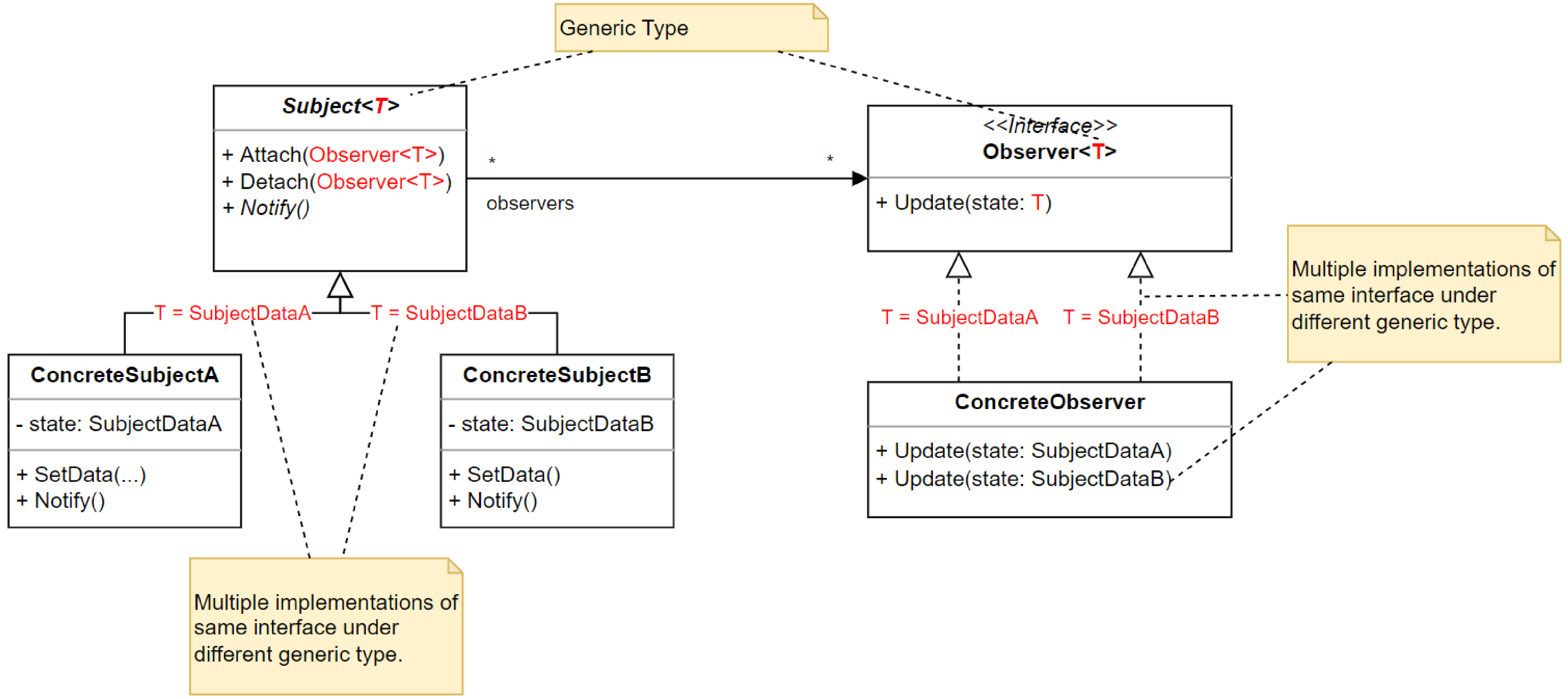


Diagram Source

Observer pattern in C#

- The Observer pattern is not in the standard library for C#
- But it is very easy to make generic interfaces and classes in C#
  - just add <T> after the name
  - And use T everywhere you want your type inserted
- E.g.:

```
public interface IObserver<T>
{
    void Update(T subject);
}
```

Demo: Full Example

Observer pattern in C#

- The observer pattern is built into C#
- With `event` and delegates a similar but more flexible mechanism is part of the language
- But that is a topic for another course

Questions?