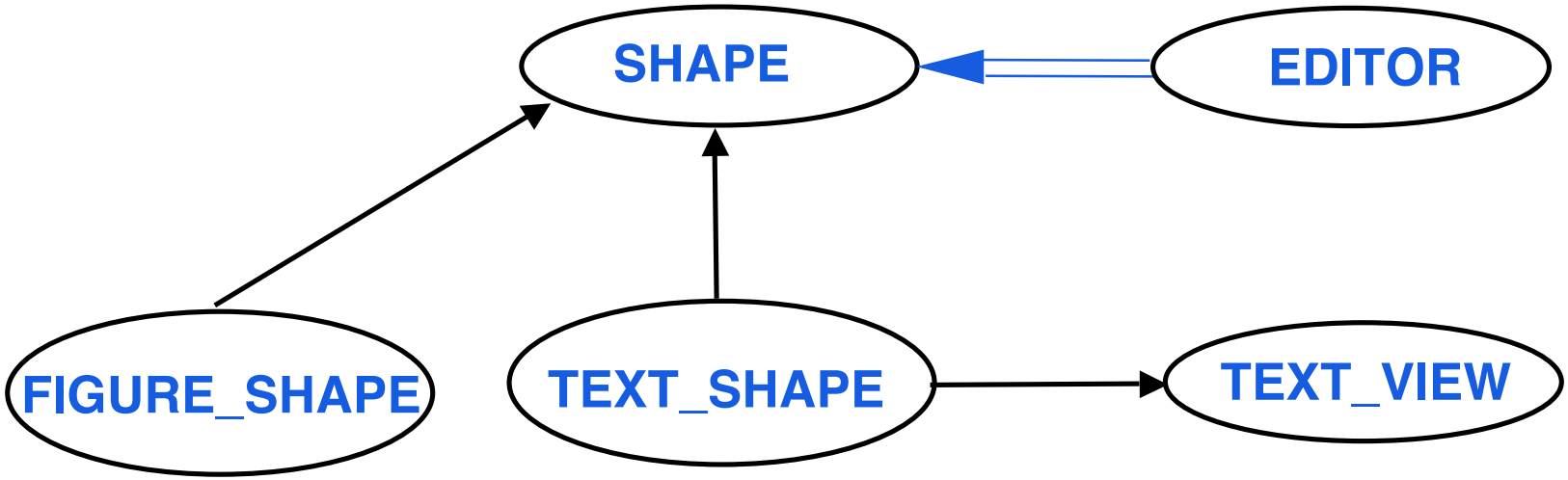
Adapter Pattern – Structural

- Intent
  - » **Convert the interface of a class into another interface that the client expects.**
  - » **Lets classes work together that couldn't otherwise**

Class Adapter – Motivation

- EDITOR expects a SHAPE
- TEXT_VIEW is not a SHAPE
- TEXT_SHAPE is a SHAPE

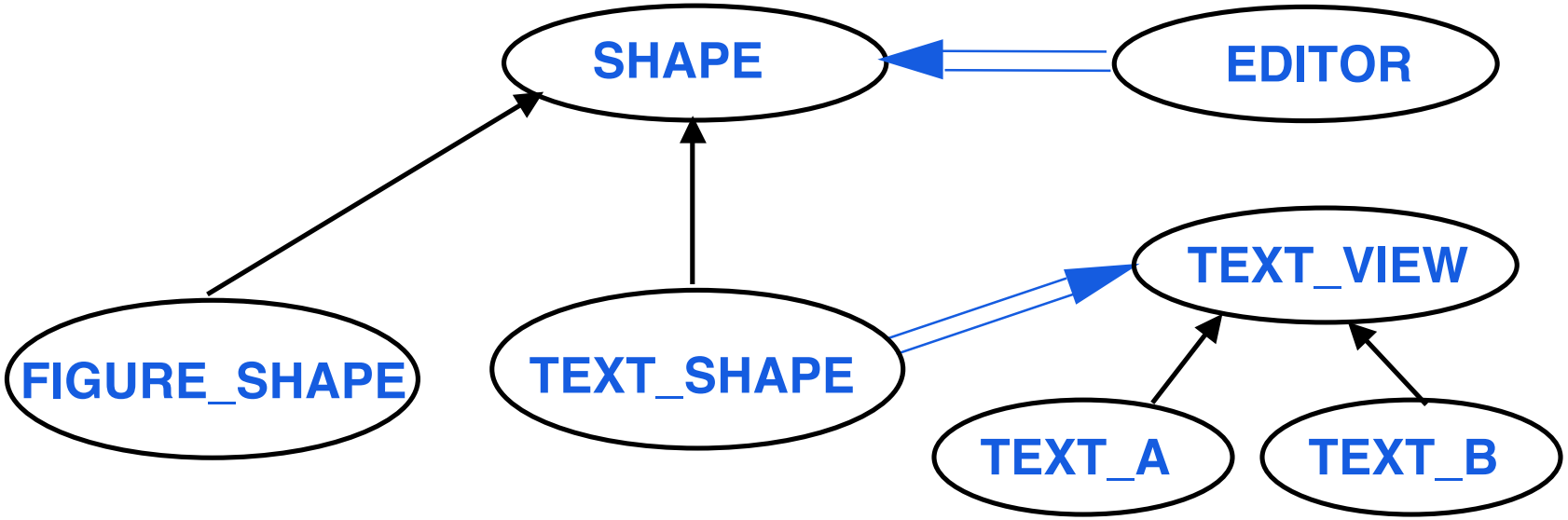
Reuse TEXT_VIEW in the context of a SHAPE



Object Adapter – Motivation

- EDITOR expects a SHAPE
- TEXT_VIEW is not a SHAPE
- TEXT_SHAPE is a SHAPE

Reuse subclasses of TEXT_VIEW in the context of a SHAPE



Adapter – Applicability

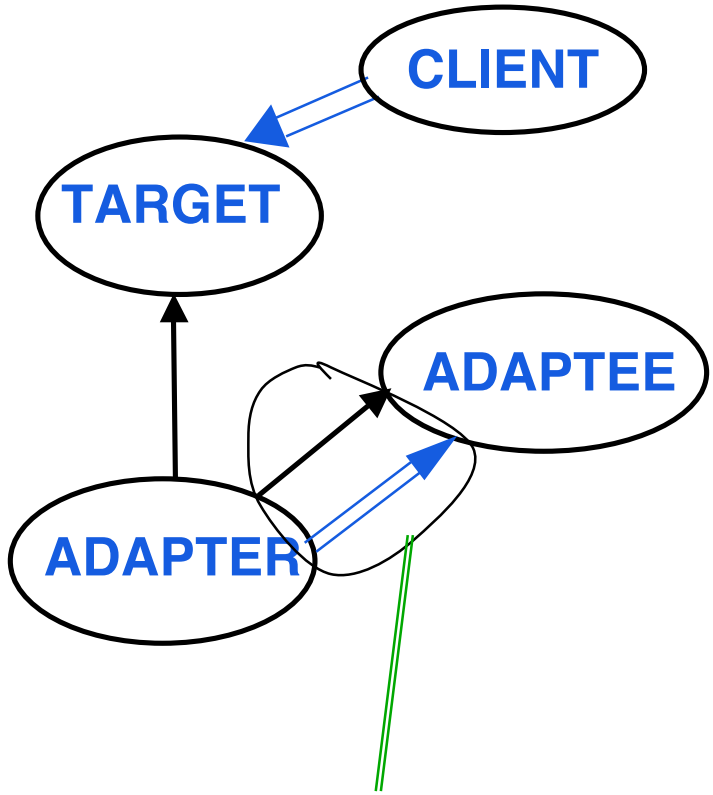
- Use an existing class when its interface does not match the one you need
- Create a class that cooperates with unrelated or unforeseen classes with incompatible interfaces
- Object Adapter Only

  Need to use several existing subclasses, but it is impractical to adapt by sub-classing each one of them

  Object adapter adapts the interface of the parent class

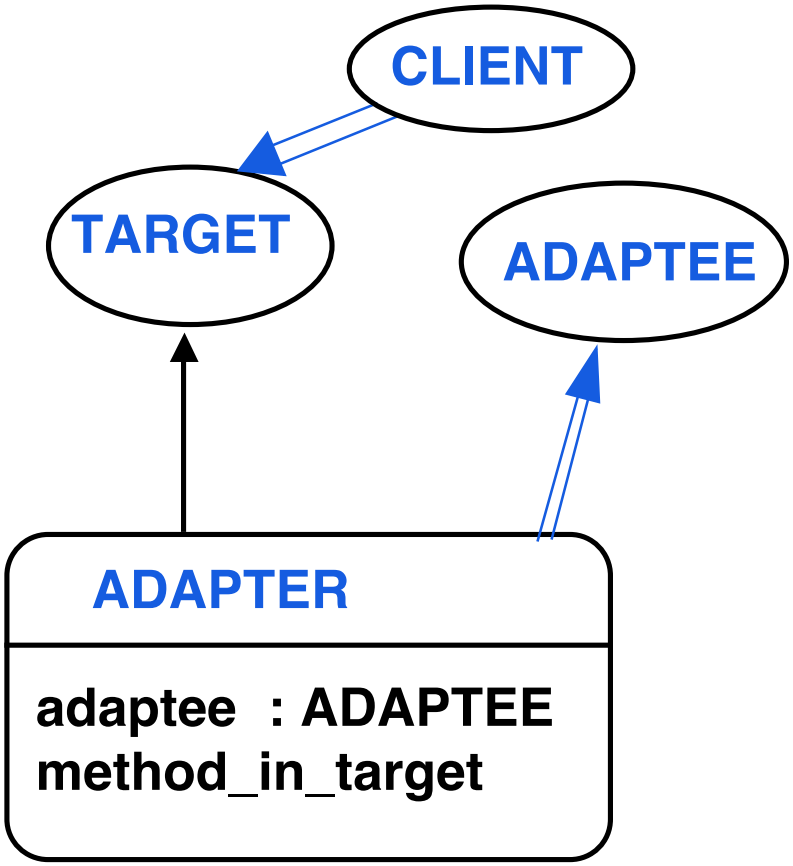
Adapter – Participants

- TARGET
  Application interface
- Client
  Target user
- Adaptee
  Interface that needs adapting
- Adapter
  – alternative name **wrapper**
  Provides functionality not provided by the adaptee



Either relationship (only 1 of them for an ADAPTER)
is_a – class adapter
has_a – object adapter

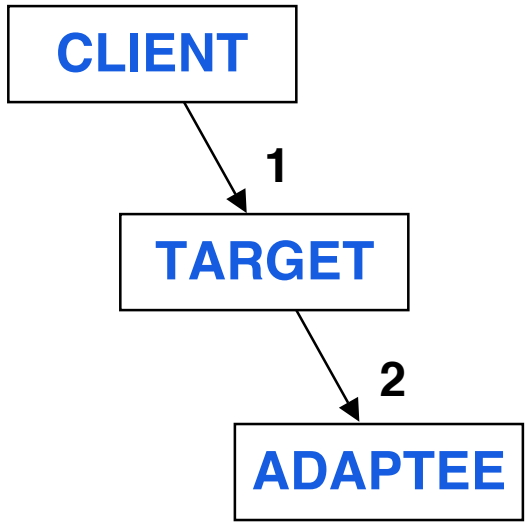
Object Adapter – Scenario



Scenario – collaboration

1 Client does
target.method_in_target

2 Adapter does
adaptee.method_in_adaptee
(note polymorphism)



Adapter Object – Pseudocode

```
class ADAPTER
   feature
       adaptee : ADAPTEE
       ...
       method_in_target is
              pre_actions
              adaptee . method_in_adaptee
              post_actions
       end
   end
end
```

```
class ADAPTEE
  feature
     method_in_adaptee is ... end
  end
end
```

Adapter Object – Stack Implementation

- The Adapter pattern used where stack operations are calls to "equivalent" sequence operations

```
Sequence container;
```

- Push – Uses the Sequence putHead

```
public void push(final Object obj) {
   container.putHead(obj);   }
```

- Pop – Use the Sequence takeHead

```
public Object pop() {
      return container.takeHead();  }
```

Adapter – Consequences

- There are tradeoffs – a class adapter – inheritance
  - » **adapts Adaptee to Target by committing to concrete Adapter class**
    - **Class adapter is not useful when we want to adapt a class and all its subclasses**
  - » **Lets Adapter override some of Adaptee's behaviour**
    - **Adapter is a subclass of Adaptee**
  - » **Introduces only one object**
    - **No additional pointer indirection is needed to get to adaptee**

Adapter – Consequences – 2

- There are tradeoffs – an object adapter – uses
  - » **One Adapter can work with many Adaptees**
    - > **Adaptee and all its subclasses**
    - > **Can add functionality to all Adaptees at once**
  - » **Makes it harder to override Adaptee behaviour**

    Requires making ADAPTER refer to the subclass rather than the ADAPTEE itself

    Or

    Subclassing ADAPTER for each ADAPTEE subclass

Adapter – Related Patterns

- Bridge is similar to object Adapter but Bridge is meant to separate interface from implementation so they can vary independently, while Adapter extends the interface of an existing object
- Decorator is more transparent than Adapter, so Decorator supports recursive composition, while Adapter doesn't
- Proxy defines a representative for another object and does not change its interface