



COMP 303

Lecture 10

Composition III

Winter 2025

slides by Jonathan Campbell

© Instructor-generated course materials (e.g., slides, notes, assignment or exam questions, etc.) are protected by law and may not be copied or distributed in any form or in any medium without explicit permission of the instructor. Note that infringements of copyright can be subject to follow up by the University under the Code of Student Conduct and Disciplinary Procedures.

Announcements

- Proposals: minor, major changes.
- Reviews: one week from today.
- Project: menu interface.
- Surveys for teams of 3.
- One-on-one meetings with TAs.

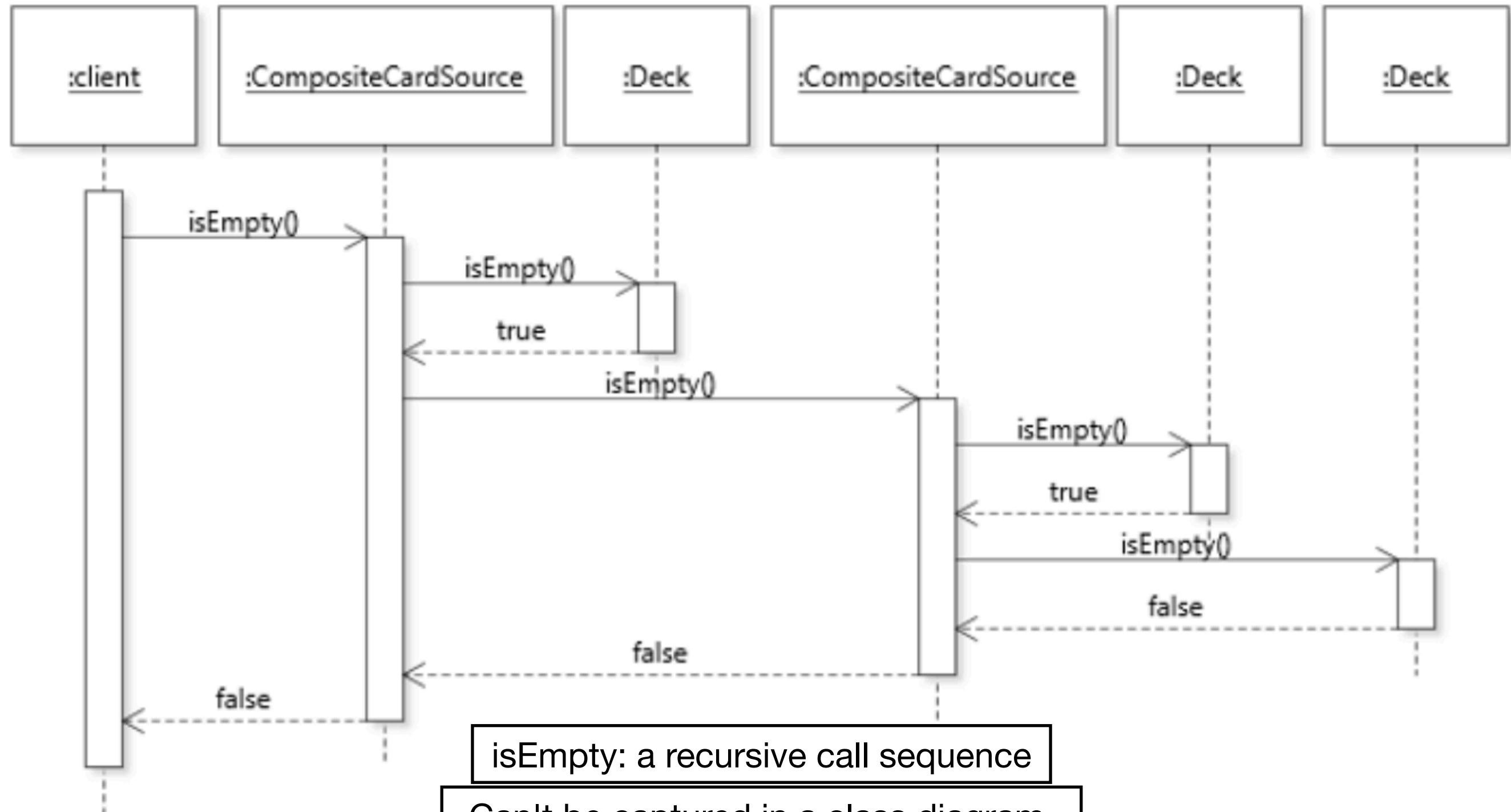
Composition

- Composite pattern & sequence diagrams
- Decorator pattern & polymorphic copying
- **Prototype & Command patterns and Law of Demeter**

Recap

isEmpty sequence diagram

Objects listed at the top, with most informative type names.

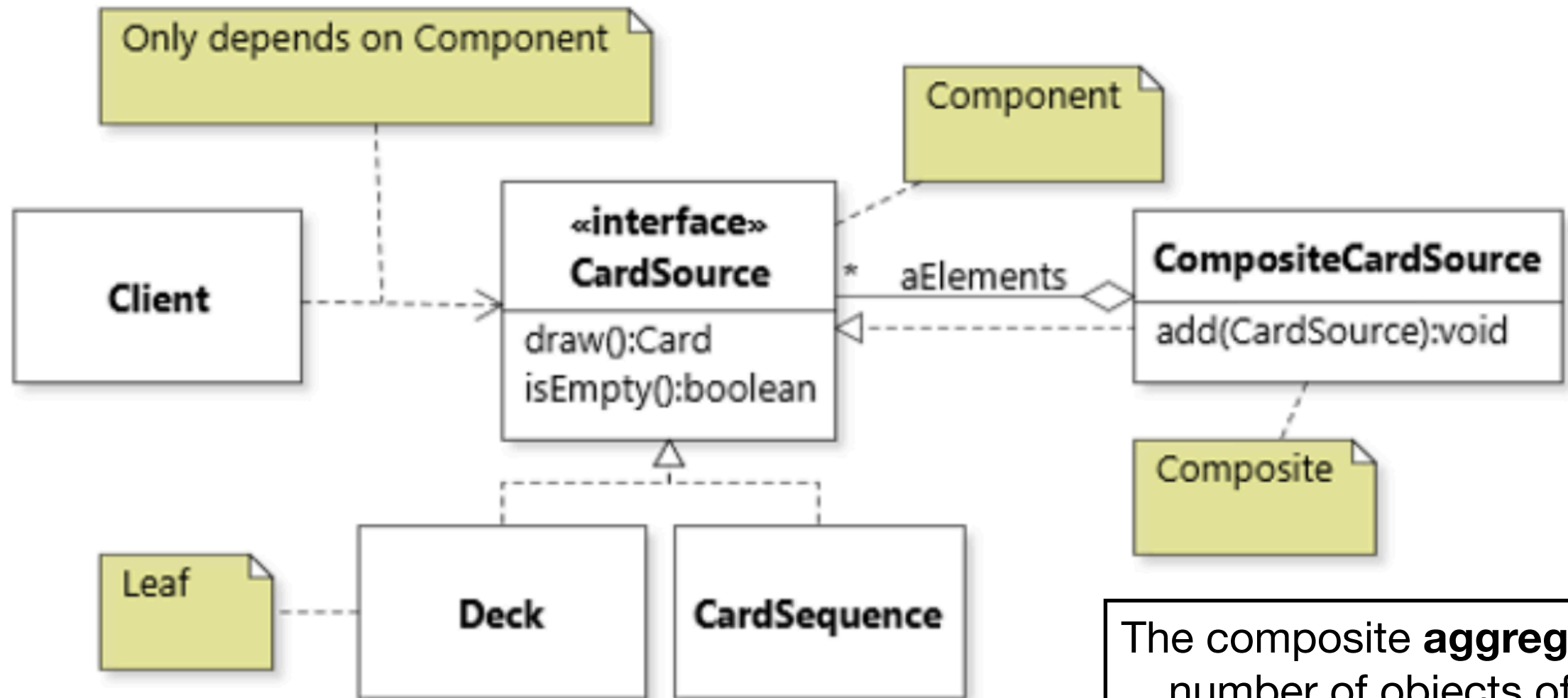


isEmpty: a recursive call sequence

Can't be captured in a class diagram.

COMPOSITE pattern

Client code should only depend on the component interface.



The composite **aggregates** a number of objects of the component type.

The composite also implements the component interface, so that it can be treated the same as any leaf.

Adding functionality to a class

- Consider that we have a class and want to *optionally* add some functionality to it. To do so, we have a few options:
 - We could use inheritance to write a child class that inherits from the base class, and implement our new functionality.
(E.g., LoggingDeck inherits from Deck.)
 - If we have an interface, we can write another implementing class.
(E.g., both Deck and LoggingDeck can implement CardSource.)

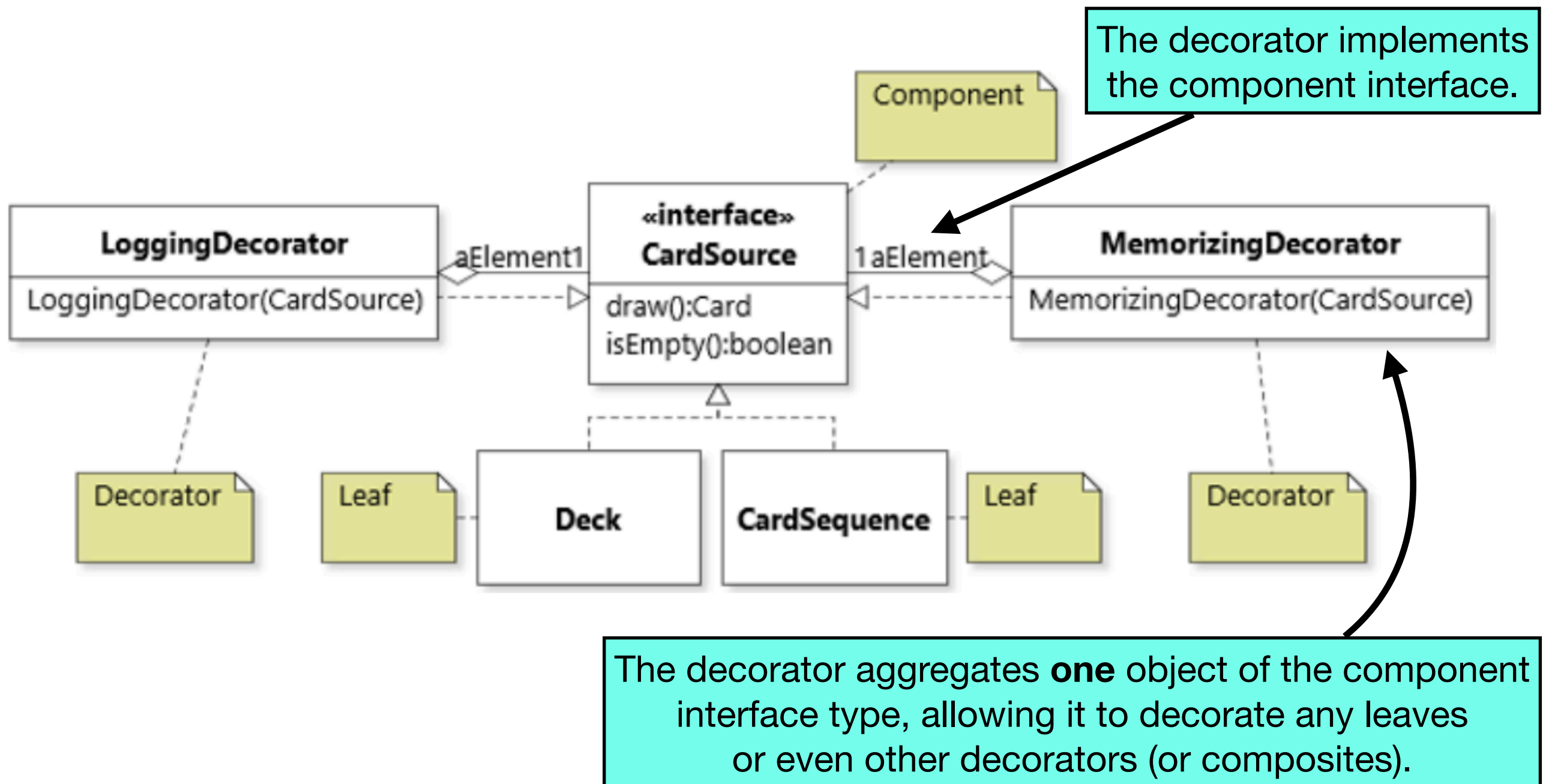
Multi-mode class

```
public class MultiModeDeck implements CardSource {
    enum Mode {
        SIMPLE, LOGGING, MEMORIZING, LOGGING_MEMORIZING
    }
    private Mode aMode = Mode.SIMPLE;
    public void setMode(Mode pMode) { ... }
    public Card draw() {
        if (aMode == Mode.SIMPLE) { ... }
        else if (aMode == Mode.LOGGING) { ... }
        ...
    }
}
```


Solution #2: DECORATOR pattern

- Context: We want to "decorate" some objects with additional functionality, while still treating those objects like any other object of the undecorated type.

DECORATOR pattern



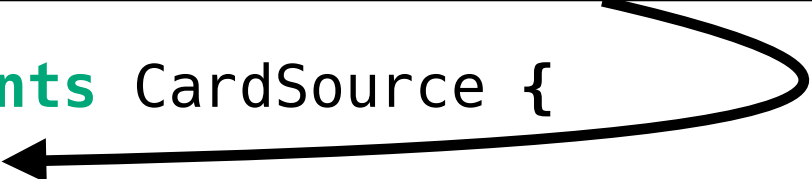
DECORATOR vs COMPOSITE

- Both the Decorator and Composite patterns feature a class that implements the component interface, and aggregates an object of the component interface type.
- But their purpose is different:
 - Composite structures objects into tree hierarchies, to treat a group of objects the same as a single instance.
 - Decorators dynamically add responsibilities to a single object, to extend behaviour without modifying the original object.

DECORATOR pattern

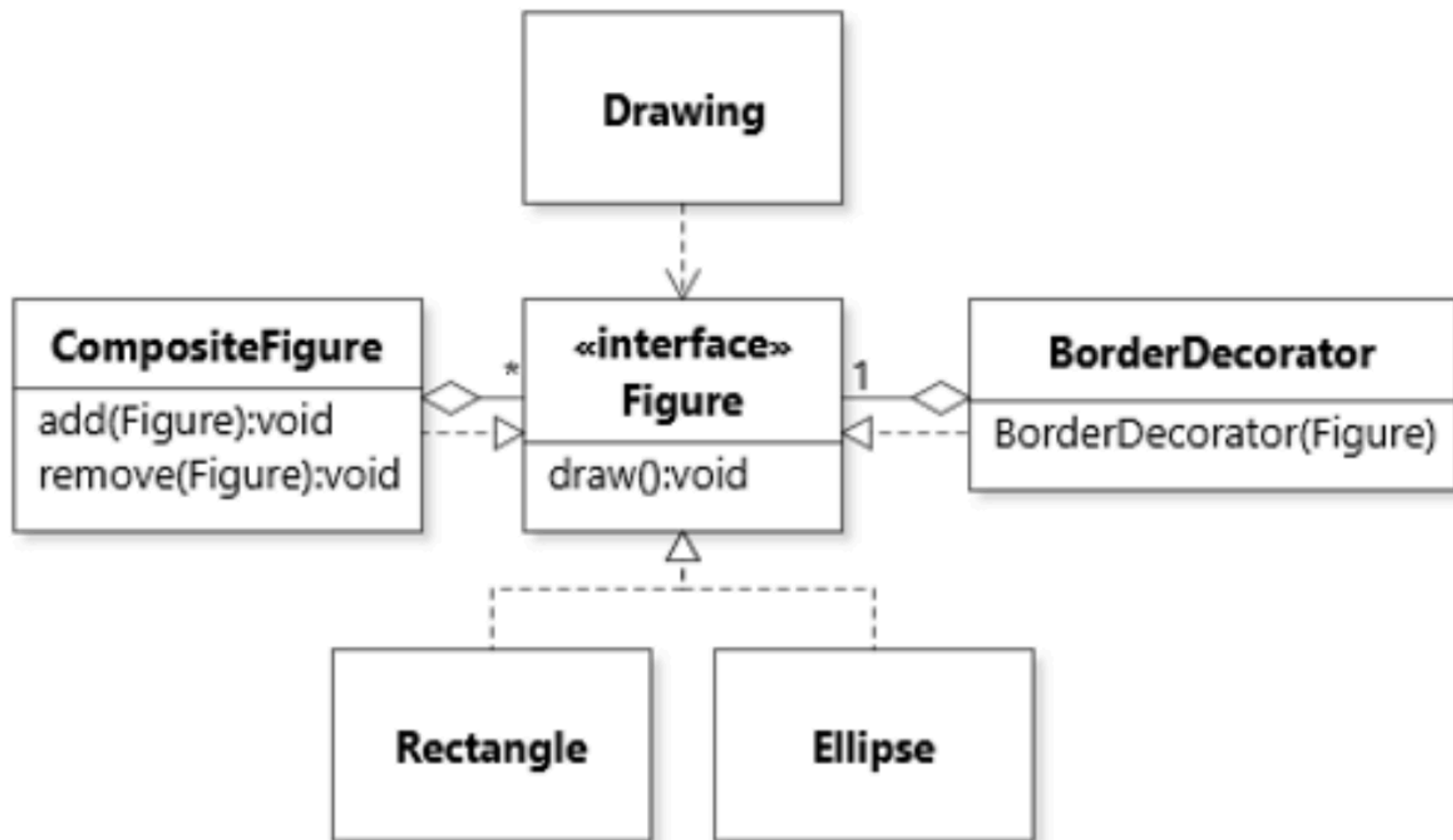
Important to set the component field (aElement) to final, because we don't want to suddenly start decorating a different object.

```
public class MemorizingDecorator implements CardSource {  
    private final CardSource aElement;  
    private final List<Card> aDrawnCards = new ArrayList<>();  
    public MemorizingDecorator(CardSource pCardSource) {  
        aElement = pCardSource;  
    }  
    public boolean isEmpty() {  
        return aElement.isEmpty();  
    }  
    public Card draw() {  
        Card card = aElement.draw(); // delegate to decorated object  
        aDrawnCards.add(card); // implement the decoration  
        return card;  
    }  
}
```



Composite + Decorator

The classic scenario: a picture drawing app



Options for removing decorators

- Maintain a reference to the undecorated component. Then re-decorate it with only the decorators you want.
- Write an unwrapping mechanism. E.g., `getWrappedObj()`. Works best if you only want to remove first.
- Field in decorator to store whether it is active.

Polymorphic copying

Polymorphic copying

- The designs that we've seen recently involve combinations of objects in elaborate object diagrams.
- One implication of this has to do with object copying.

Polymorphic copying

- We've seen that we can implement a copy constructor to make a copy. But to use such a constructor, we must specify a particular type, which can be a problem when using polymorphism:

```
List<CardSource> sources = ...;
List<CardSource> copy = new ArrayList<>();
for (CardSource source : sources) {
    copy.add(???); // which constructor to call?
}
```

Polymorphic copying

```
CardSource copy = null;
if (source.getClass() == Deck.class) {
    copy = new Deck((Deck) source);
} else if (source.getClass() == CardSequence.class) {
    copy = new CardSequence((CardSequence) source);
} else if (source.getClass() == CompositeCardSource.class) {
    copy = new CompositeCardSource((CompositeCardSource) source);
}
...
```

Voids the benefit of polymorphism, which is to work with instances of CardSource no matter what their actual concrete type is.

Also: an example of the Switch Statement anti-pattern.

Also: The CompositeCardSource copy constructor would need to have the same pattern.

Polymorphic copying

- Polymorphic copying: Make copies of objects without knowing the concrete type of the object.

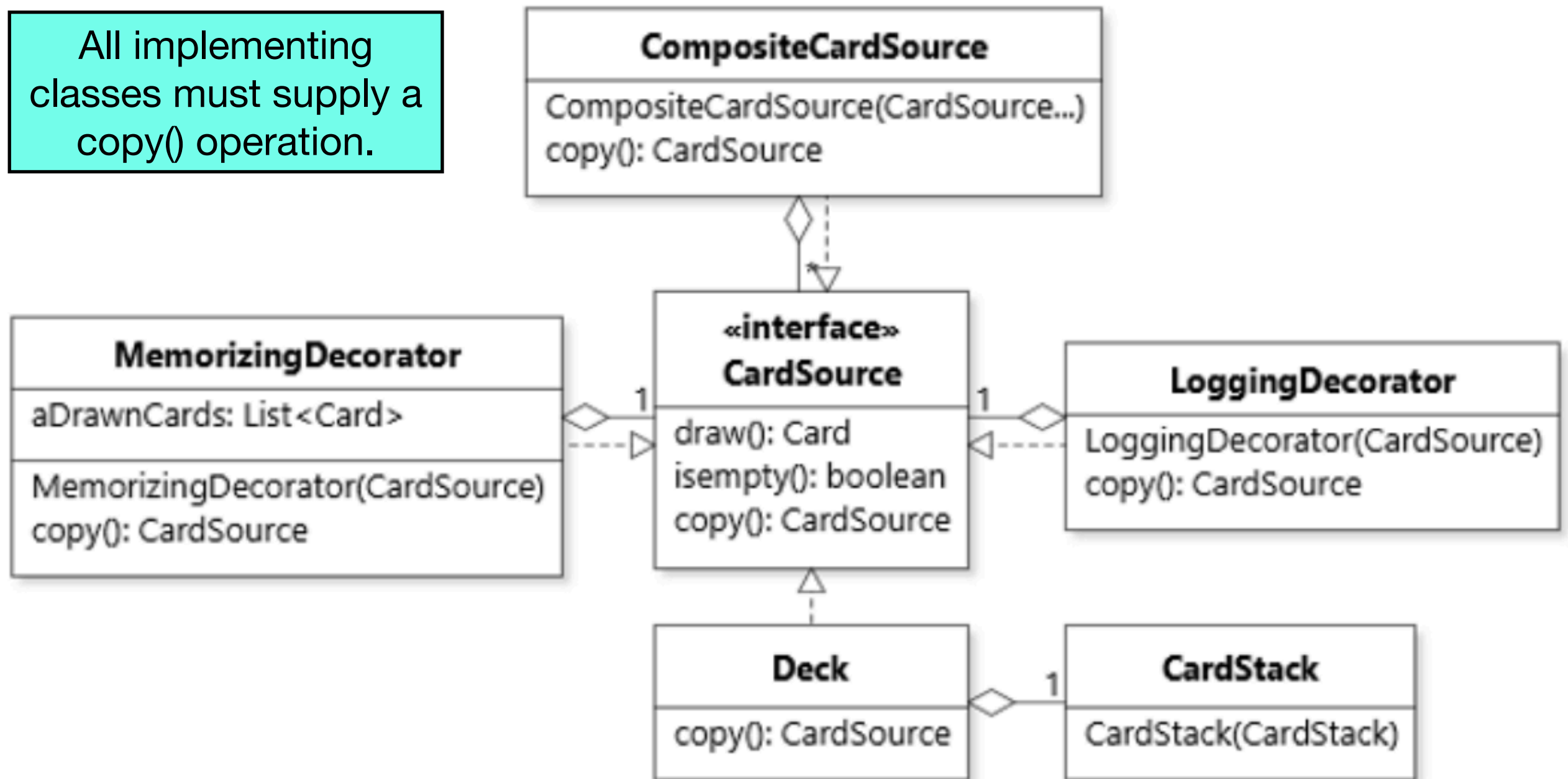
Polymorphic copying

```
public interface CardSource {  
    ...  
  
    /**  
     * @return An object that is an exact deep copy  
     * (distinct object graph) of this card source.  
     */  
    CardSource copy();  
}
```

Now all implementing classes must specify a copy() operation.

Polymorphic copying

All implementing classes must supply a `copy()` operation.



Implementing copy()

```
public class Deck implements CardSource {  
    private CardStack aCards = new CardStack();  
    ...  
    public Deck copy() {  
        Deck copy = new Deck();  
  
        // copy all of its state  
        copy.aCards = new CardStack(aCards);  
  
        return copy;  
    }  
}
```

Implementing copy()

```
public class LoggingDecorator implements CardSource {
    private CardSource aSource;
    public LoggingDecorator (CardSource pSource) {
        aSource = pSource;
    }
    ...
    public LoggingDecorator copy() {
        // call the copy constructor when possible
        // and call copy() method on the decorated object
        return new LoggingDecorator(aSource.copy());
    }
}
```

Implementing copy()

```
public class CompositeCardSource implements CardSource {  
    public CardSource copy() {  
        CompositeCardSource copy = new CompositeCardSource();  
  
        copy.aElements = new ArrayList<>();  
        for(CardSource source : aElements) {  
            copy.aElements.add(source.copy());  
        }  
  
        return copy;  
    }  
}
```


PROTOTYPE pattern

Polymorphic copying

```
List<CardSource> sources = ...;  
List<CardSource> copy = new ArrayList<>();  
for (CardSource source : sources) {  
    copy.add(source.copy());  
}
```

Polymorphic instantiation

```
public class GameModel {  
    private CardSource aCardSource;  
    public void newGame() {  
        aCardSource = /* Instantiate a new CardSource */;  
    }  
}
```

Same problem as with copying a CardSource:
which constructor do we call?

Polymorphic instantiation

```
public class GameModel {  
    private CardSource aCardSource;  
    public void newGame() {  
        aCardSource = new Deck();  
    }  
}
```

What if we want to configure GameModel to use any type of CardSource, and change the default type at **runtime** (i.e., pass an argument to the constructor)?

Would need a switch statement as before.

Solution #1: Class<T>

```
public class GameModel {  
    private CardSource source;  
    public <T extends CardSource> void newGame(Class<T> sourceCls) {  
        try {  
            source = sourceCls.getDeclaredConstructor().newInstance();  
        } catch (InstantiationException | IllegalAccessException |  
InvocationTargetException | NoSuchMethodException e) {  
            throw new RuntimeException("Error creating CardSource of type: "  
+ sourceCls.getName(), e);  
        }  
    }  
}
```

Known as **reflection**. Will see more in a later lecture.

Avoids needing to call constructor on a particular type.

But, messy: requires lots of error handling and can be fragile.

Solution 2: PROTOTYPE pattern

```
public class GameModel {  
    private final CardSource aCardSourcePrototype;  
    private CardSource aCardSource;  
    public GameModel(CardSource pCardSourcePrototype) {  
        aCardSourcePrototype = pCardSourcePrototype;  
        newGame();  
    }  
    public void newGame() {  
        aCardSource = aCardSourcePrototype.copy();  
    }  
}
```

#1: pass in prototype object to constructor

#2: call copy() on prototype object
when we would normally call constructor

Avoids needing to call constructor on a particular type; uses polymorphism.

PROTOTYPE pattern

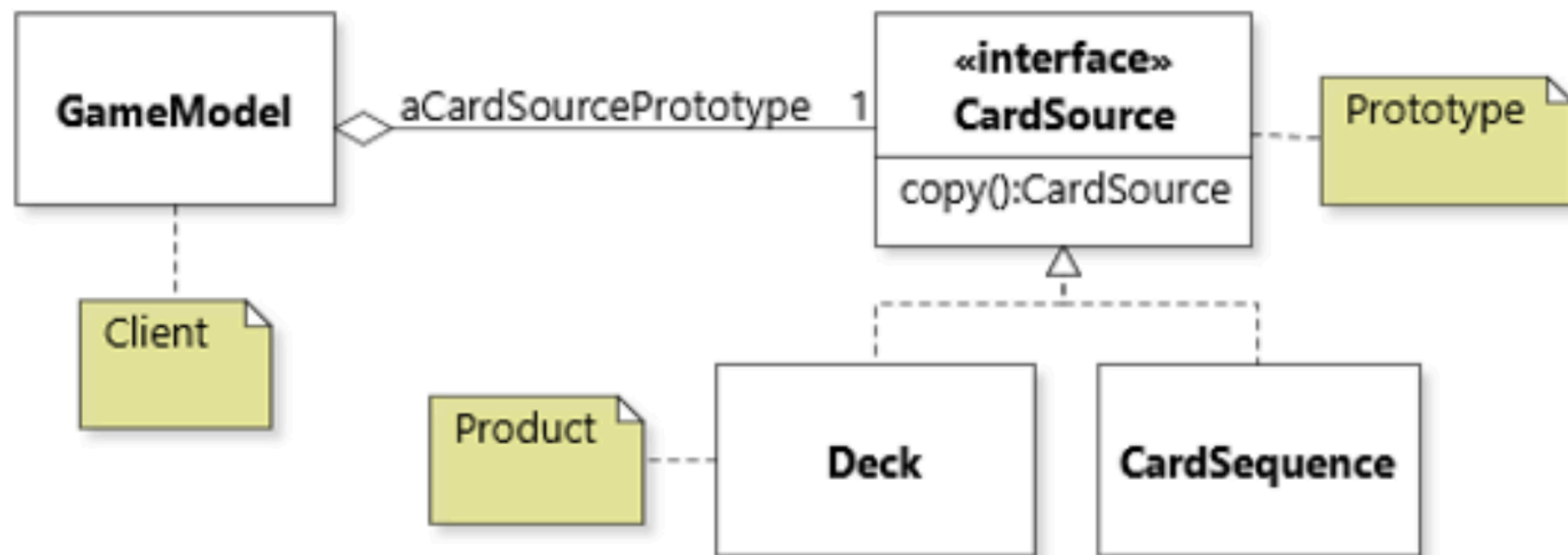
- Used when we have to create objects whose type may not be known at compile time.
- Involves storing a reference to a prototype object, then polymorphically copying that object whenever a new instance is required.
- By calling `copy()`, we don't need a large switch statement or chain of conditional statements; we simply make a copy of whatever the current prototype is, regardless of its type (since all subtypes of the particular interface have `copy` defined).

Benefits of the PROTOTYPE pattern

- Polymorphic instantiation: we can clone an object without needing to know its concrete class.
- If it takes a long time or a lot of effort to initialize an object but not much time or effort to copy it, and we like to make a lot of these objects, then it's easier to make it once and clone it as necessary instead.
 - You could even maintain a registry of prototypes (e.g., a dictionary, like the Flyweight pattern), each with a different configuration.

PROTOTYPE pattern

Prototype: the *abstract* element (typically an interface) whose *concrete* prototype must be instantiated at runtime.



Products: the objects that can be created by copying the prototype.

COMMAND pattern

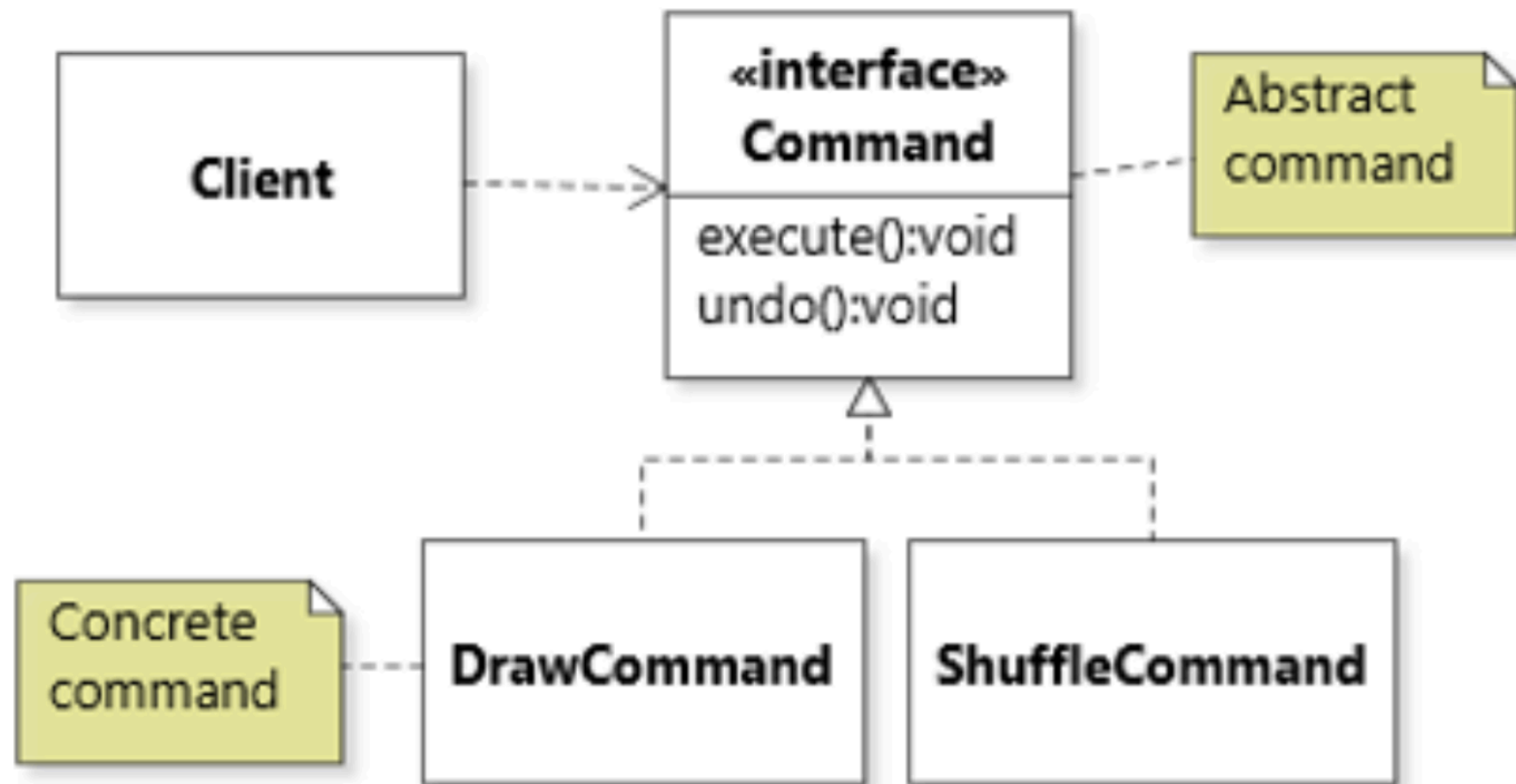
Commands

- Command: a piece of code for a cohesive action: e.g., saving a file to disk, drawing a card from a deck, etc.
 - We typically represent each action as its own method or function, e.g., `shuffle()` or `draw()`.
- Defining a piece of functionality as its own method is great. But sometimes we want to go further, and define a piece of functionality as its own *object*.

Commands

- In particular, for these kinds of reasons:
 - we want to store a history of commands that have been executed
 - we want to undo or replay commands
 - we want to accumulate commands and then execute them all at once

COMMAND pattern



Each piece of functionality is defined in its own class, which implements a Command interface.

Draw command

```
public class Deck {  
    private CardStack aCards = new CardStack();  
    public Command createDrawCommand() {  
        return new Command() {  
            Card aDrawn = null;  
            public Optional<Card> execute() {  
                aDrawn = draw();  
                return Optional.of(aDrawn);  
            }  
            public void undo() {  
                aCards.push(aDrawn);  
                aDrawn = null;  
            }  
        };  
    }  
}
```

Factory method

Returns anonymous class.

Provides implementations of execute and undo.

Anonymous classes retain a reference to their outer instance, so we can access draw() and aDrawn.

```
Deck deck = new Deck();  
Command command = deck.createDrawCommand();  
Card card = command.execute().get();  
command.undo();
```

Design choices (1)

- Access to command target: how can the command modify the objects it acts on?
 - Do they store it as state? Is it passed in as argument? Is it defined as an anonymous inner class?
- Data flow: How is the output of the command returned?
 - Is it returned from execute? Or is the result stored as state inside the command and then accessed through a getter method?

Design choices (2)

- Command execution correctness
 - Can commands be executed more than once?
 - Do commands have specific preconditions? (Design by contract.)
- Encapsulation of target objects: How can the command affect private state of the target object?
 - Is it defined as an anonymous inner class (and thus has access to private fields)?

Design choices (3)

- Storing data
 - If the command can be undone, some state may need to be saved. Should it be stored directly in the command object, or some external structure?

Commands in the project

- Command, ChatCommand and MenuCommand.
- Design choices

References

- Robillard ch. 6.6-6.8 (p. 144-153)
 - Exercises #14-16, 18: <https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter6.md>

Coming up

- Next lecture:
 - Unit testing