McGill

COMP 303

Lecture 9

Composition II

Winter 2025

slides by Jonathan Campbell

Announcements

- codesample.info
- Brainstorming meetings
- For groups of 3: survey coming soon.
- Project complexity.
    - https://www.mcgill.ca/study/2024-2025/university_regulations_and_resources/undergraduate/gi_credit_system
- One-on-one meetings with TAs

Composition

- Composite pattern & sequence diagrams
- **Decorator pattern & polymorphic copying (today)**
- Prototype & Command patterns & the Law of Demeter

Recap

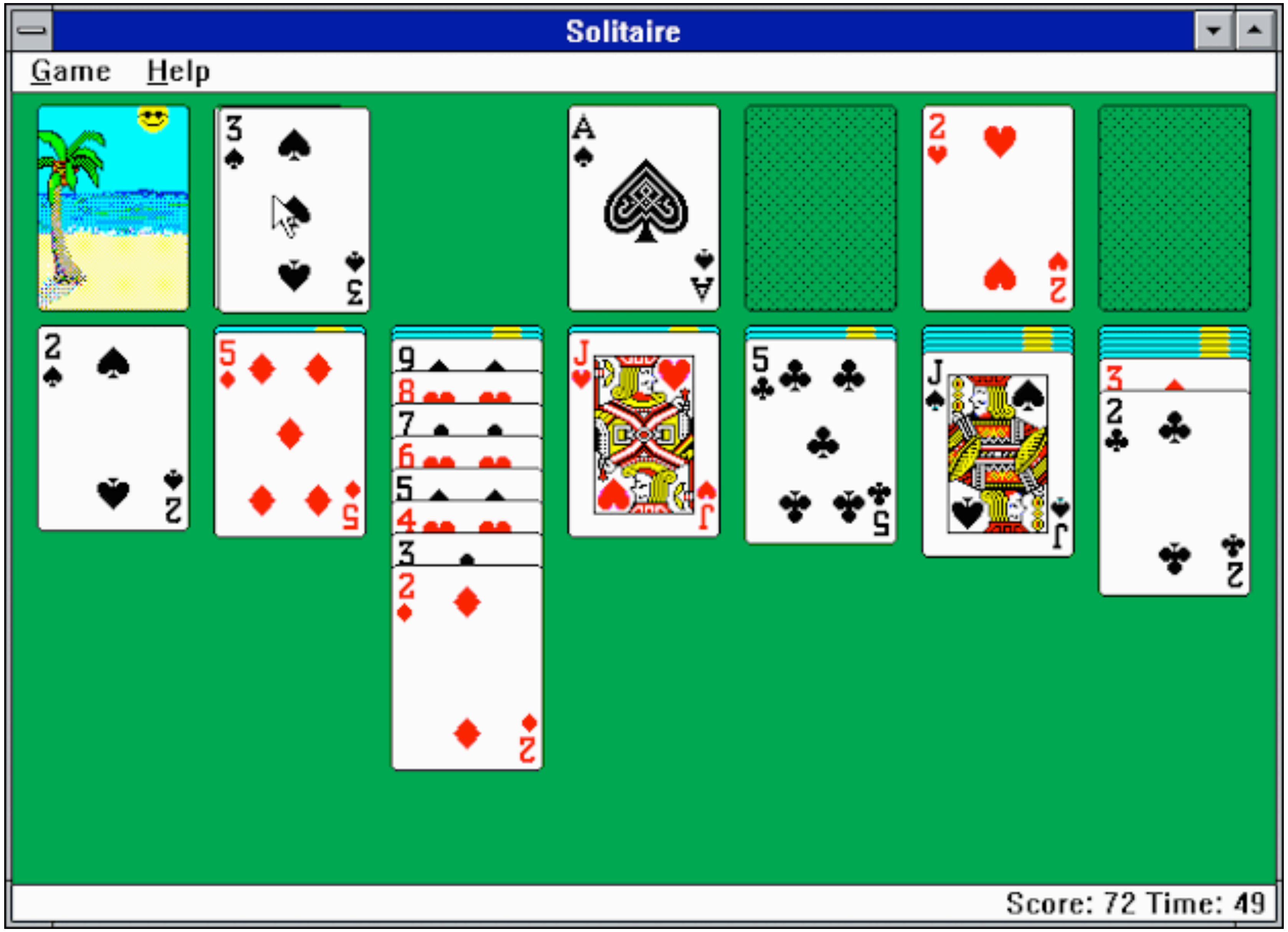
Composition

- Composition: one object holding a reference to another.
    - "Has-a" relationship: A deck "has a" number of cards.
- A very important concept: large software systems are always assembled from smaller parts, and composition is one of the main ways to do this (also, inheritance).
    - We like to design larger abstractions in terms of smaller ones.

Composition

- The solution to two common design situations:
    - Aggregation: when an abstraction must contain a collection of other abstractions. E.g., a Deck that contains ("aggregates") a collection of Cards ("components").
    - Delegation: when a class is too big (God class anti-pattern), we may want to break it down so that it contains aggregates of smaller classes, and then delegate responsibility to each part.
- These purposes are not mutually-exclusive; they can sometimes can be used together.

Example: Solitaire

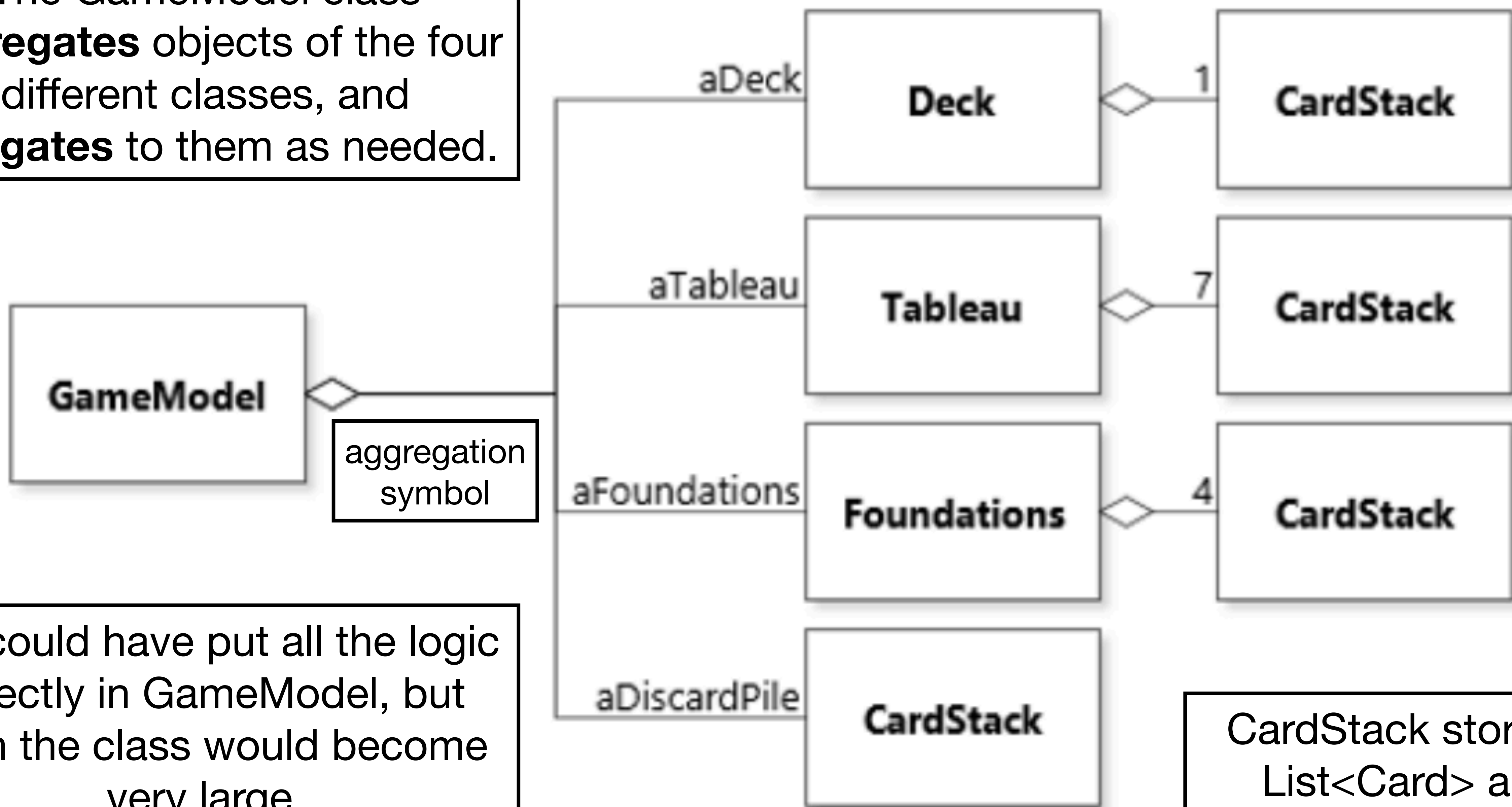


Windows 3.0 solitaire (https://bgr.com/wp-content/uploads/2015/08/windows-solitaire-30.png)

Example: Solitaire

The Tableau and Foundations **aggregate** various CardStacks.

The GameModel class **aggregates** objects of the four different classes, and **delegates** to them as needed.



We could have put all the logic directly in GameModel, but then the class would become very large.

CardStack stores a List<Card> and associated methods (push, pop, peek, etc.)

Composition

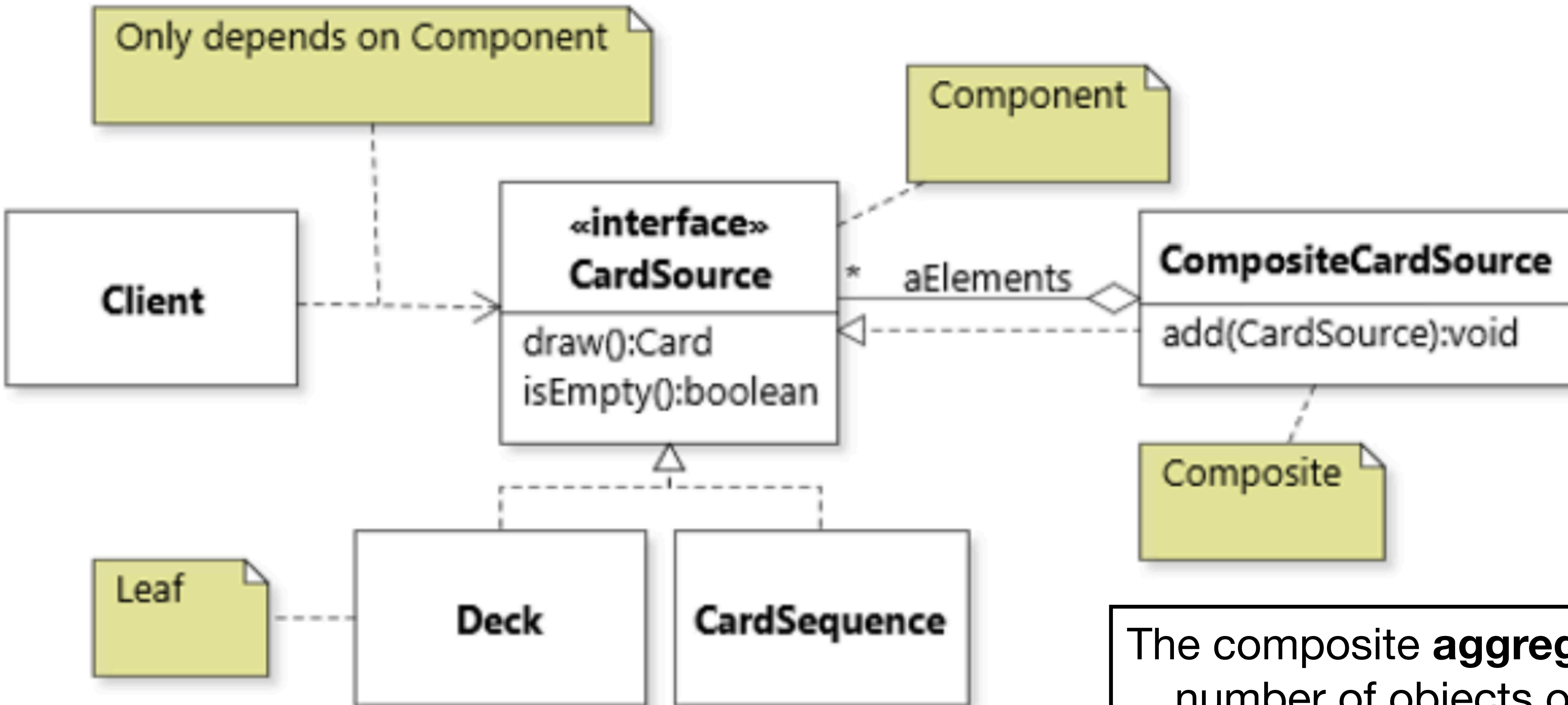
- There are specific design patterns we can use to compose objects to avoid unnecessary complications.
  - COMPOSITE pattern
  - DECORATOR pattern
  - PROTOTYPE pattern
  - COMMAND pattern

COMPOSITE pattern

- Situation: We'd like to have a group of objects behave like a single object.
    - For example: a class that aggregates a bunch of CardSources should itself be treated as a CardSource.

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.

Composite pattern examples

- All these composite classes implemented the component interface, while aggregating object(s) of said interface.
  - CompositeCardSource (CardSource)
  - Building (MapObject)
  - Mashup (Song)

Sequence diagrams

Sequence diagrams

- The use of composition involves objects collaborating with each other, i.e., objects calling methods on other objects at run-time.
  - Contrasts with static design decisions, which involve which classes depend on which other classes.
- It can be helpful to model design decisions related to object call sequences. We can do so using a sequence diagram.

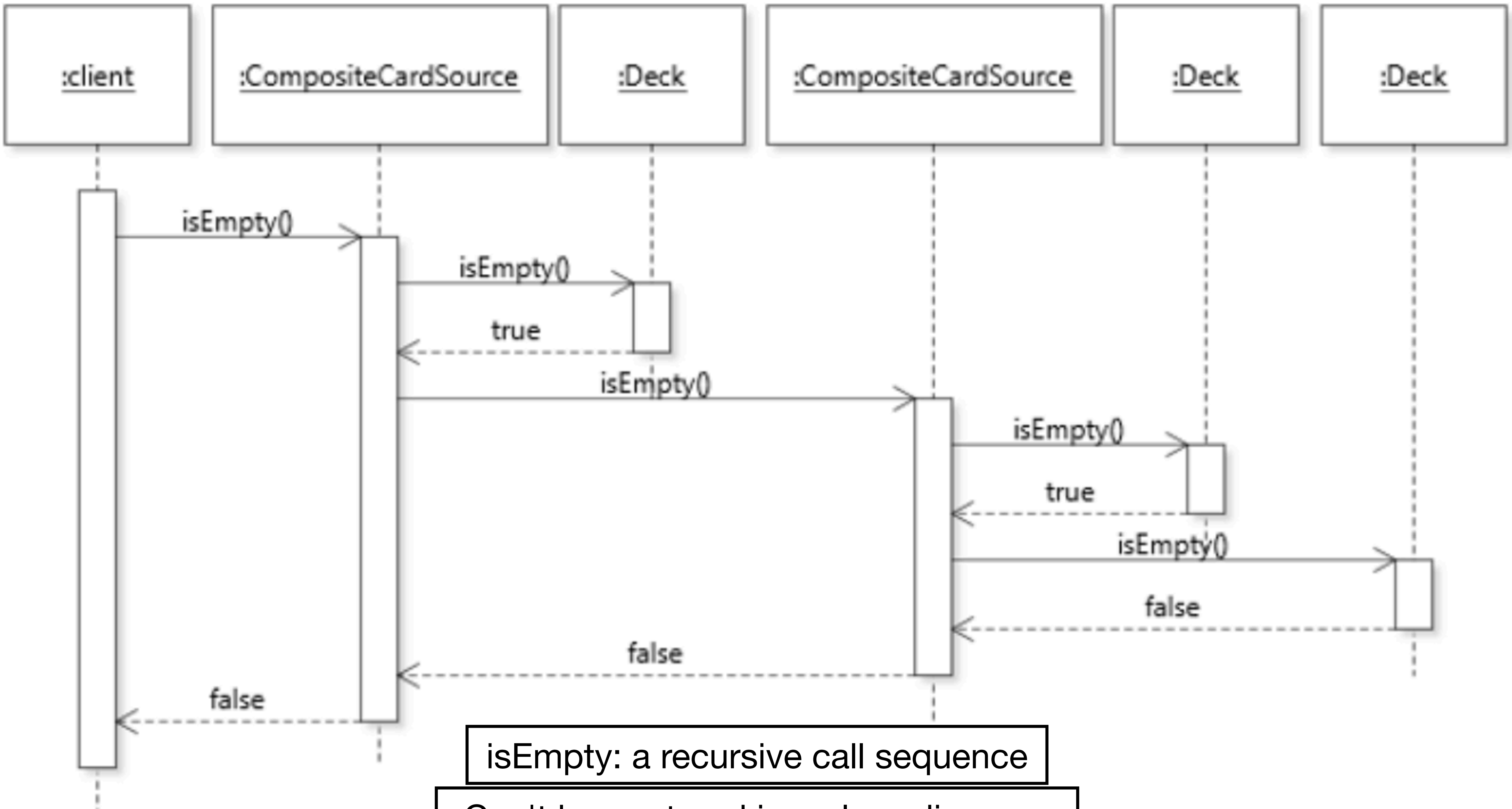
Sequence diagrams

- Assume that client code creates a CompositeCardSource object, which aggregates two CardSources: a Deck, and another CompositeCardSource, which itself contains two Decks.

- Let's model the client calling isEmpty() on its card source.

isEmpty sequence diagram

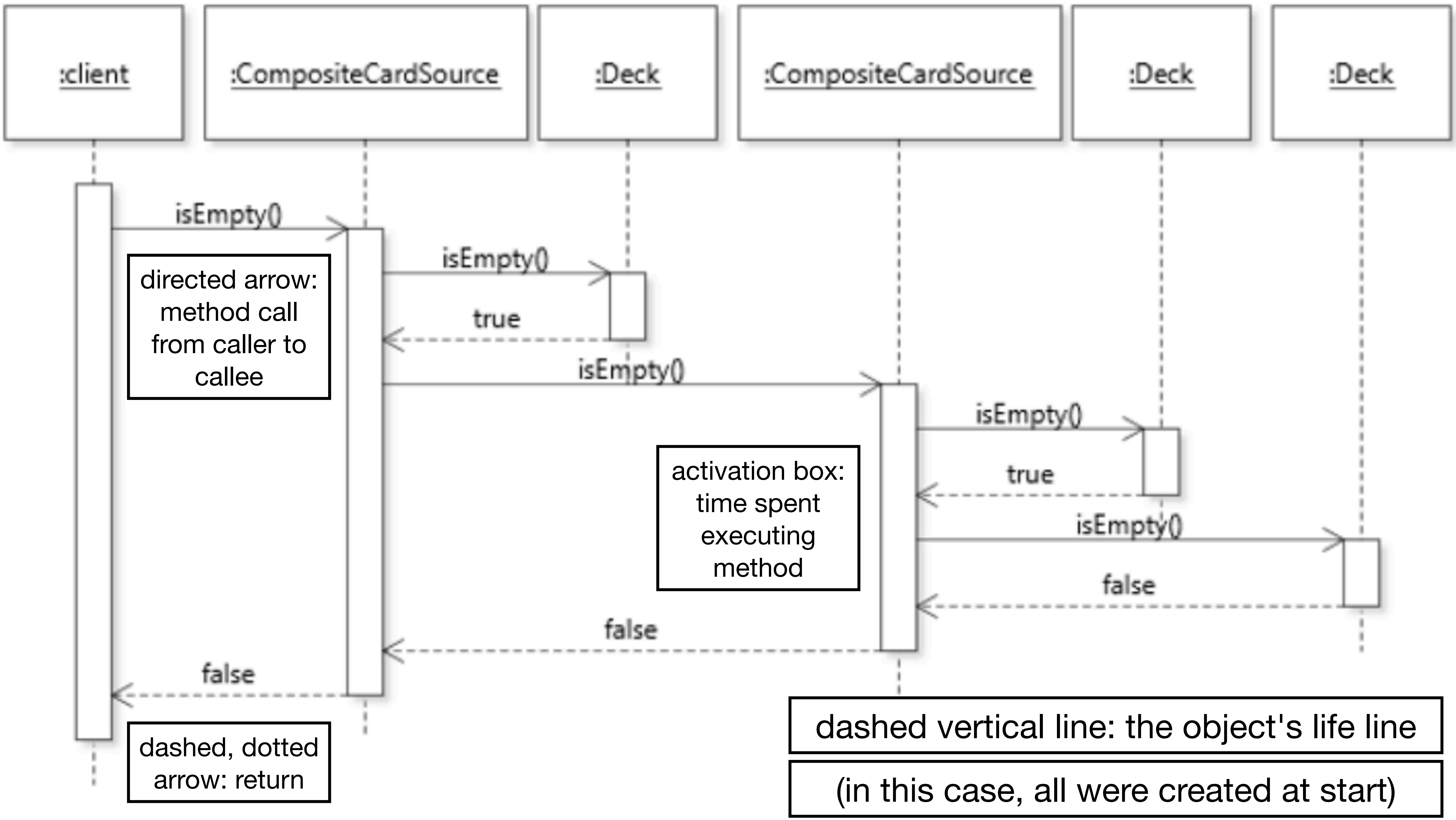
Objects listed at the top, with most informative type names.



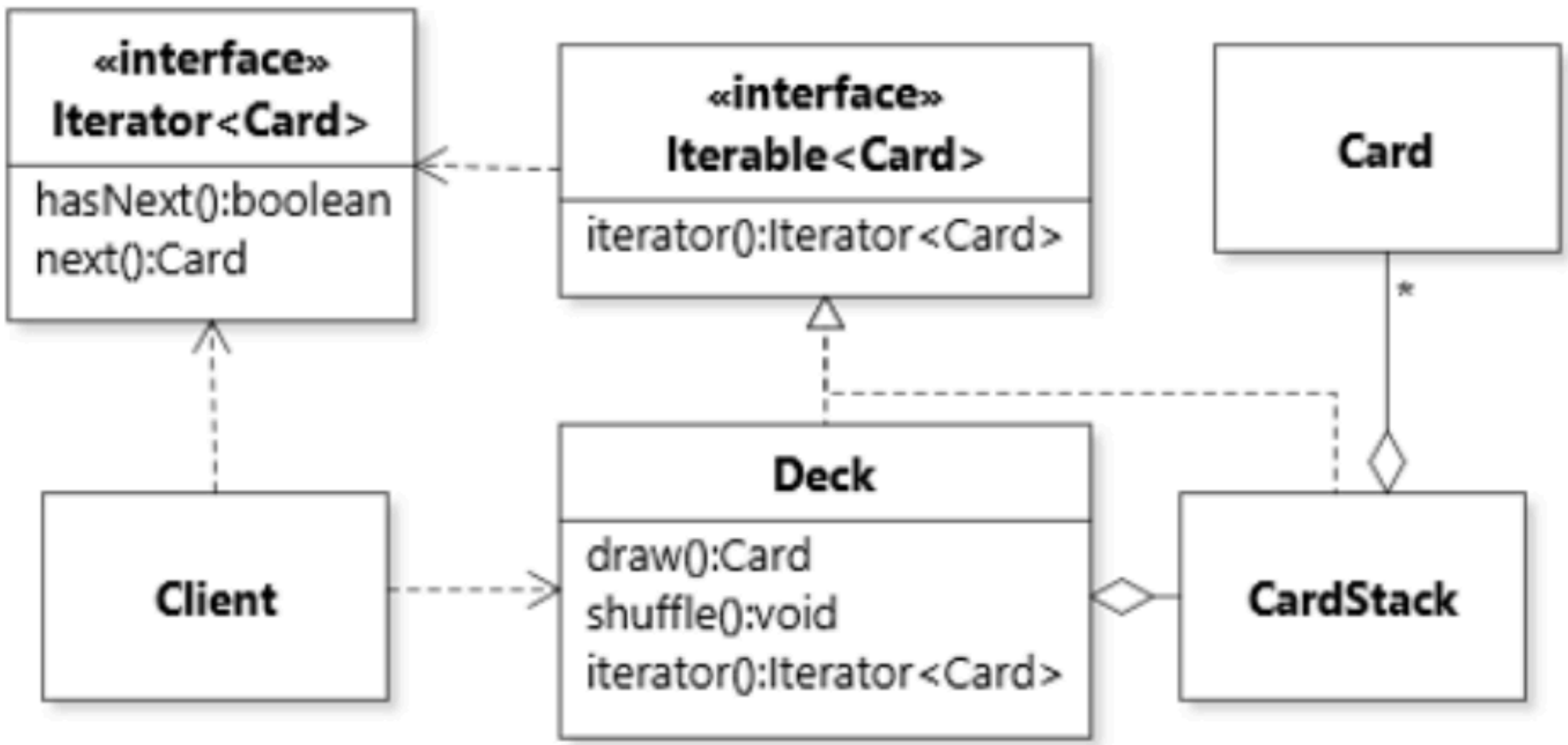
isEmpty: a recursive call sequence

Can't be captured in a class diagram.

isEmpty sequence diagram



Iterator class diagram



Deck has a CardStack.

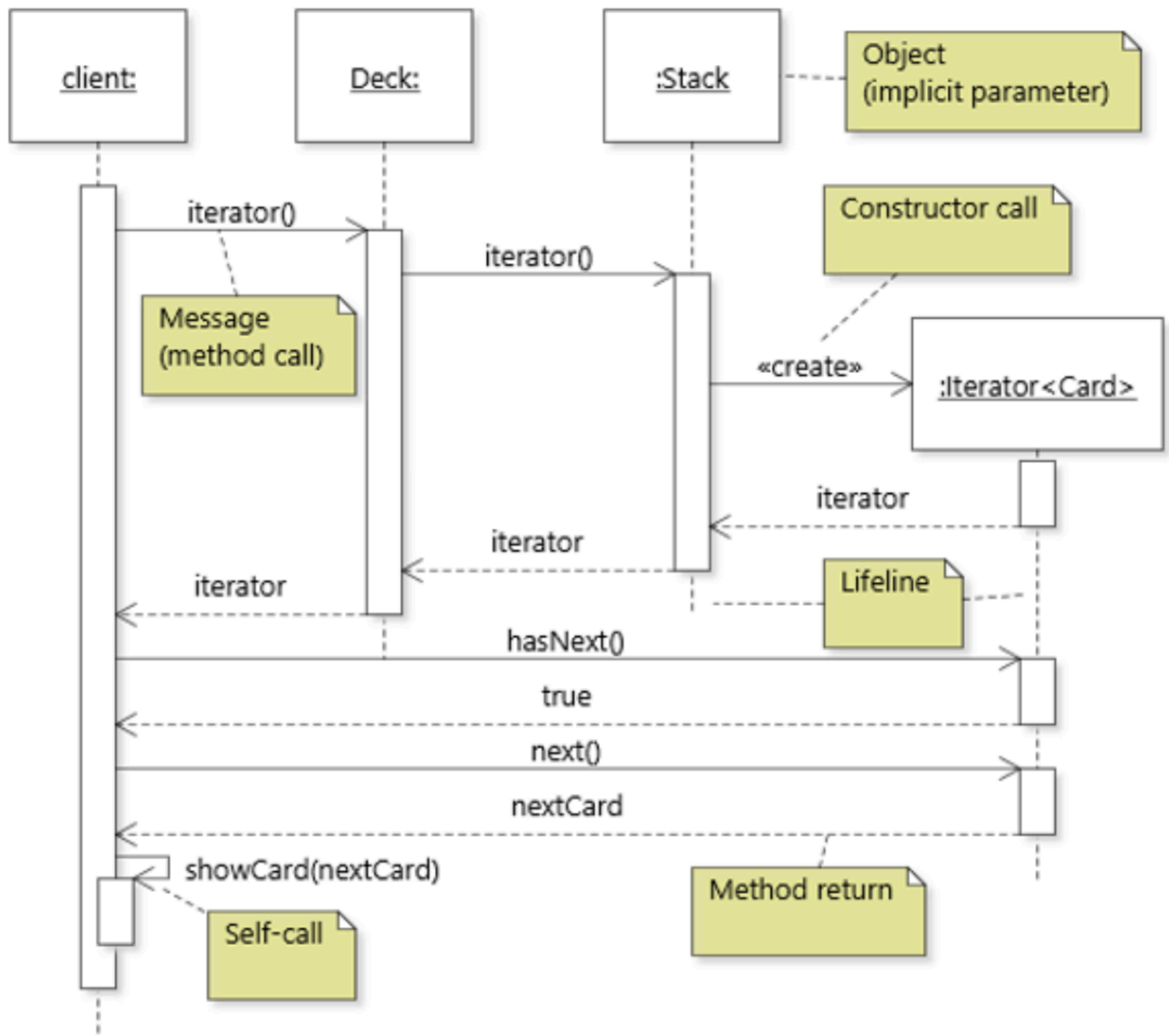
Deck and CardStack implement Iterable.

Iterator sequence diagram

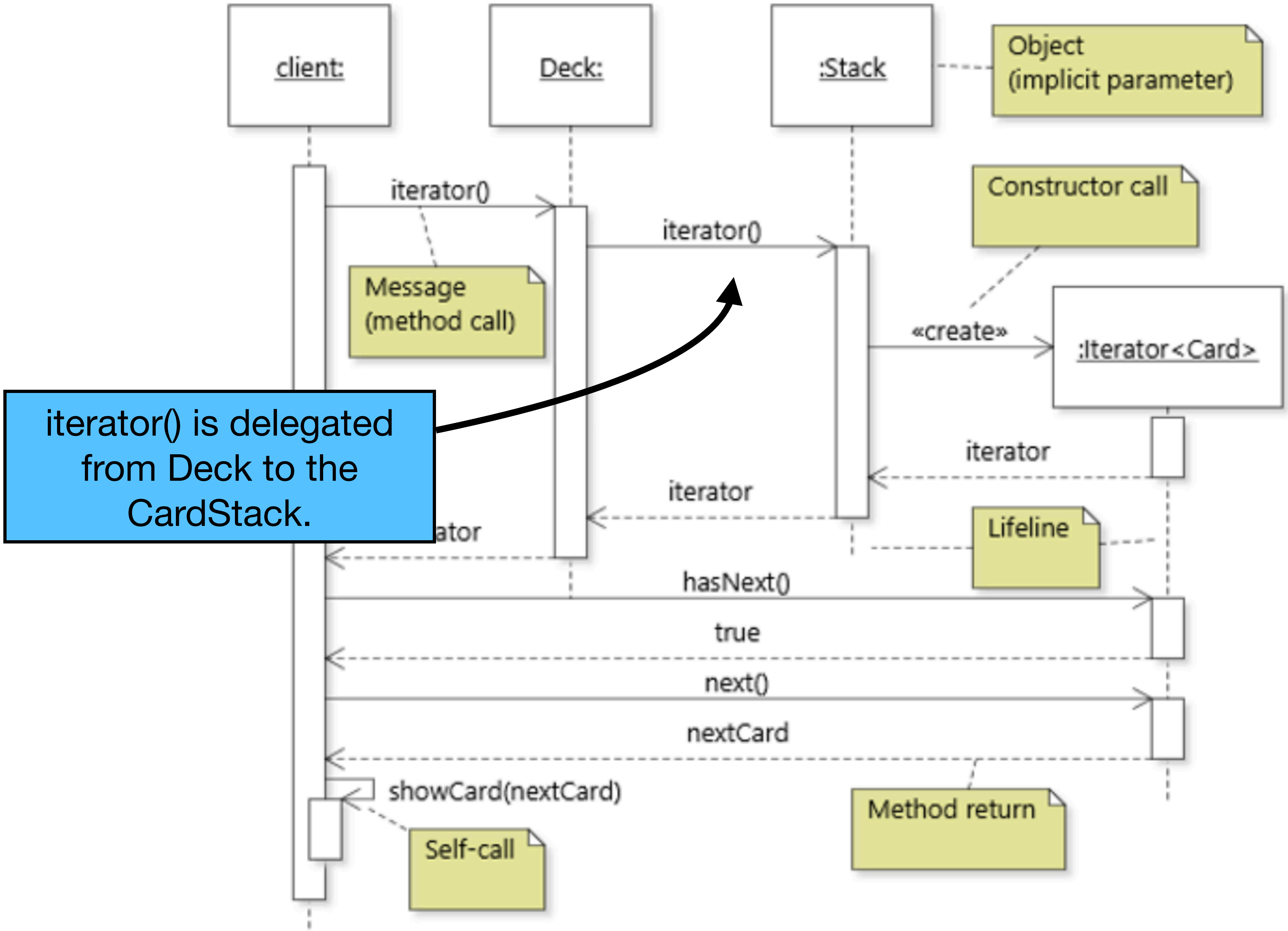
- Suppose we want to model client code like this:

```
for (Card card : Deck) {
   this.showCard(card)
}
```

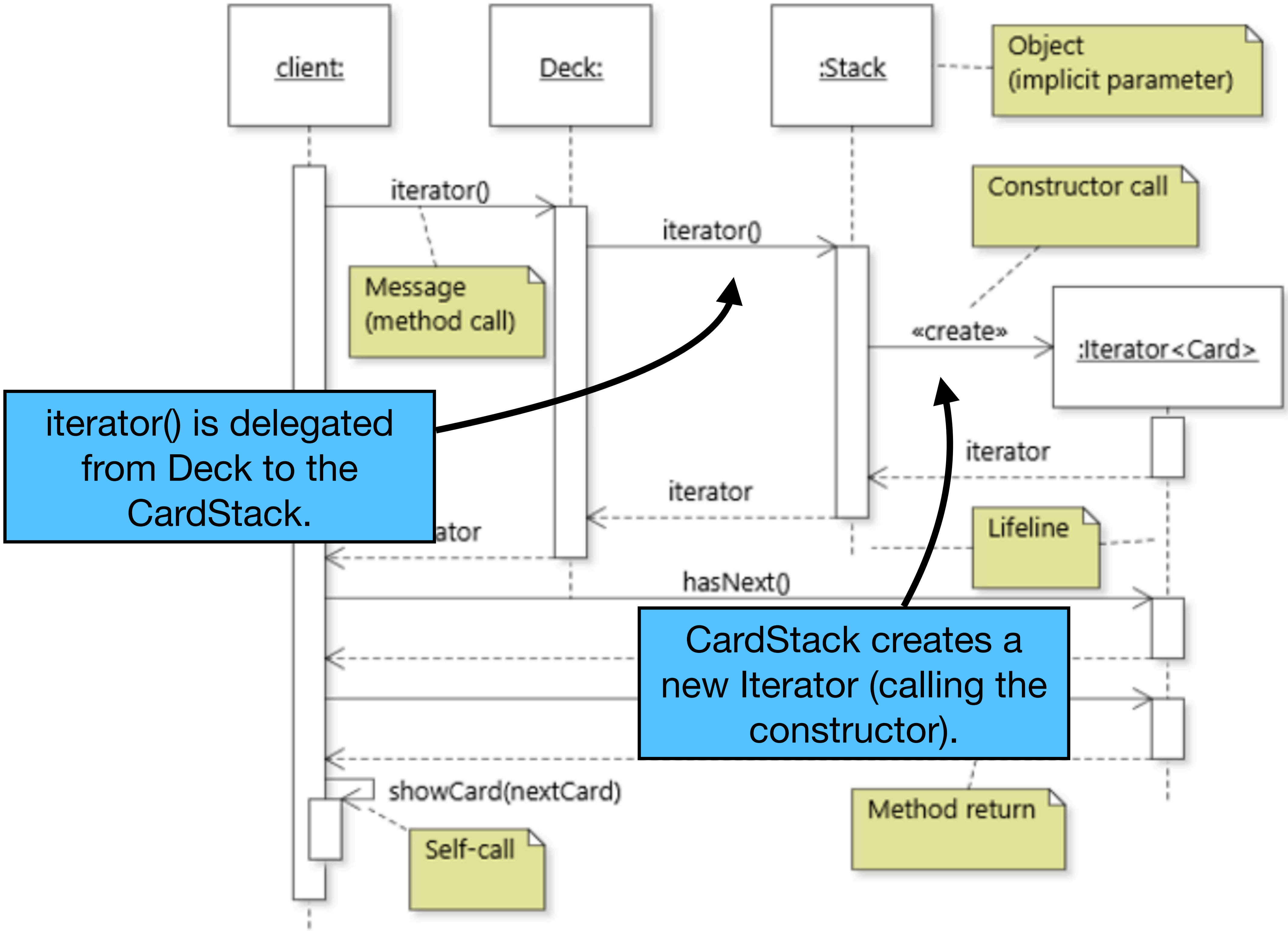
Iterator sequence diagram



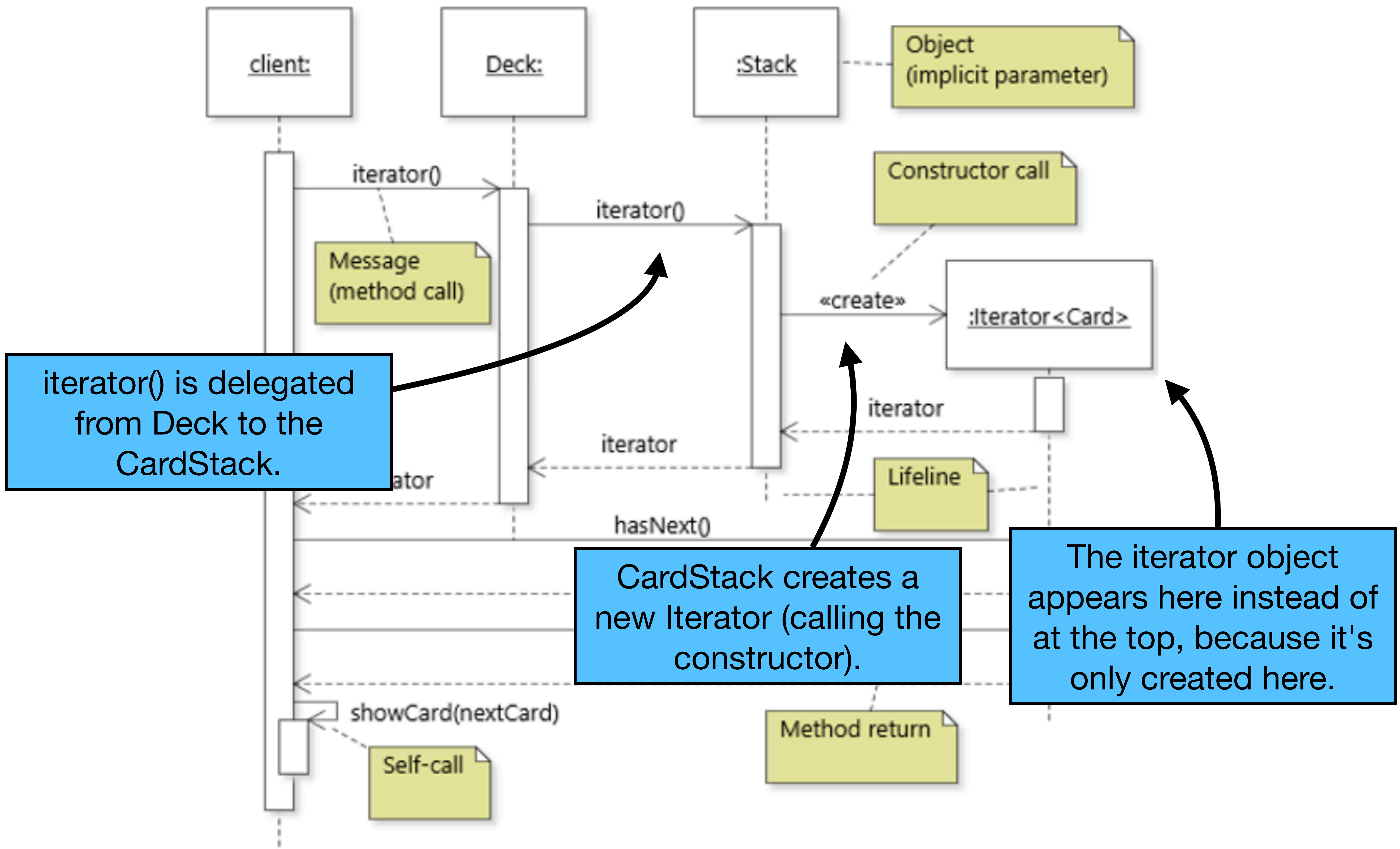
Iterator sequence diagram



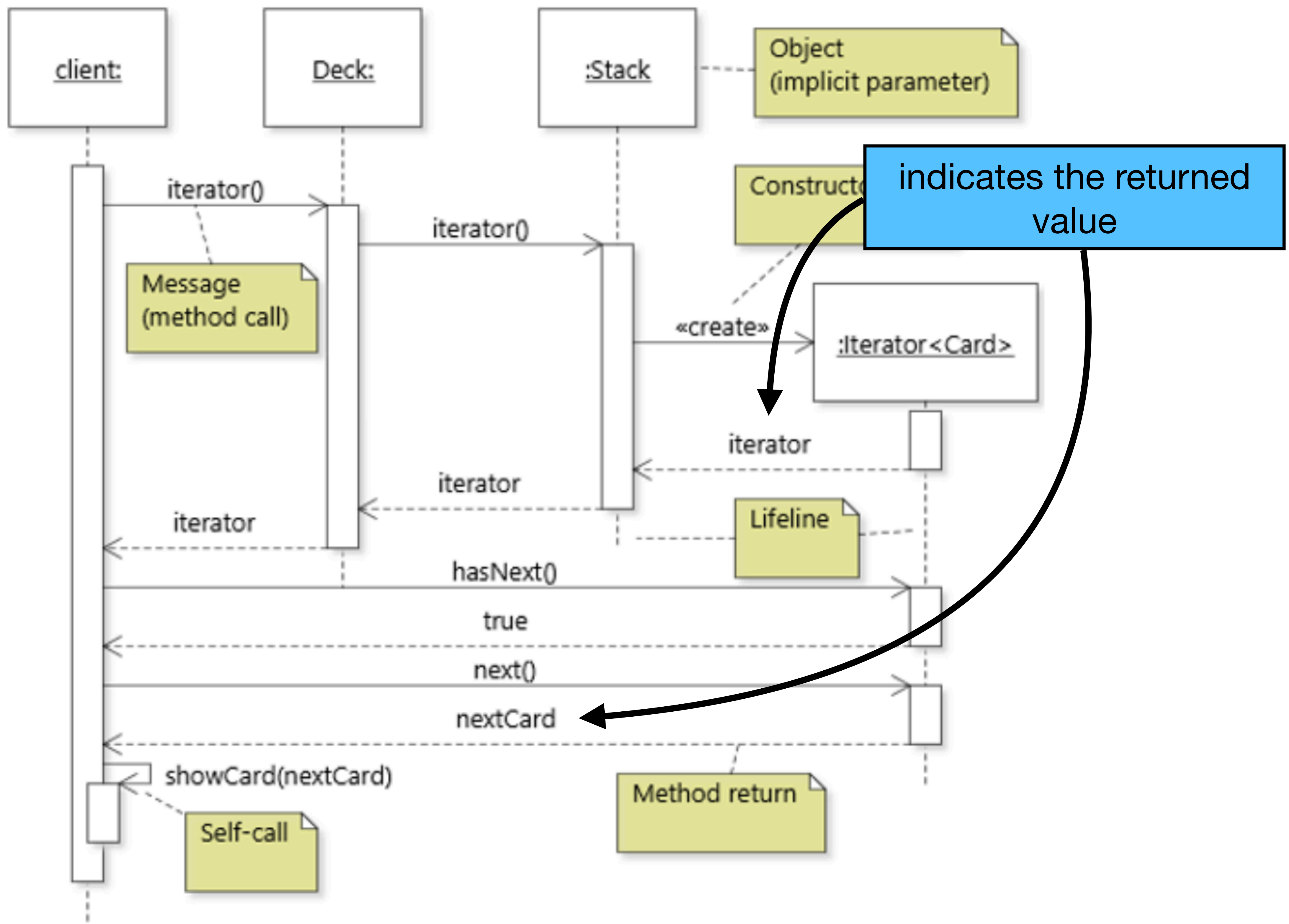
Iterator sequence diagram



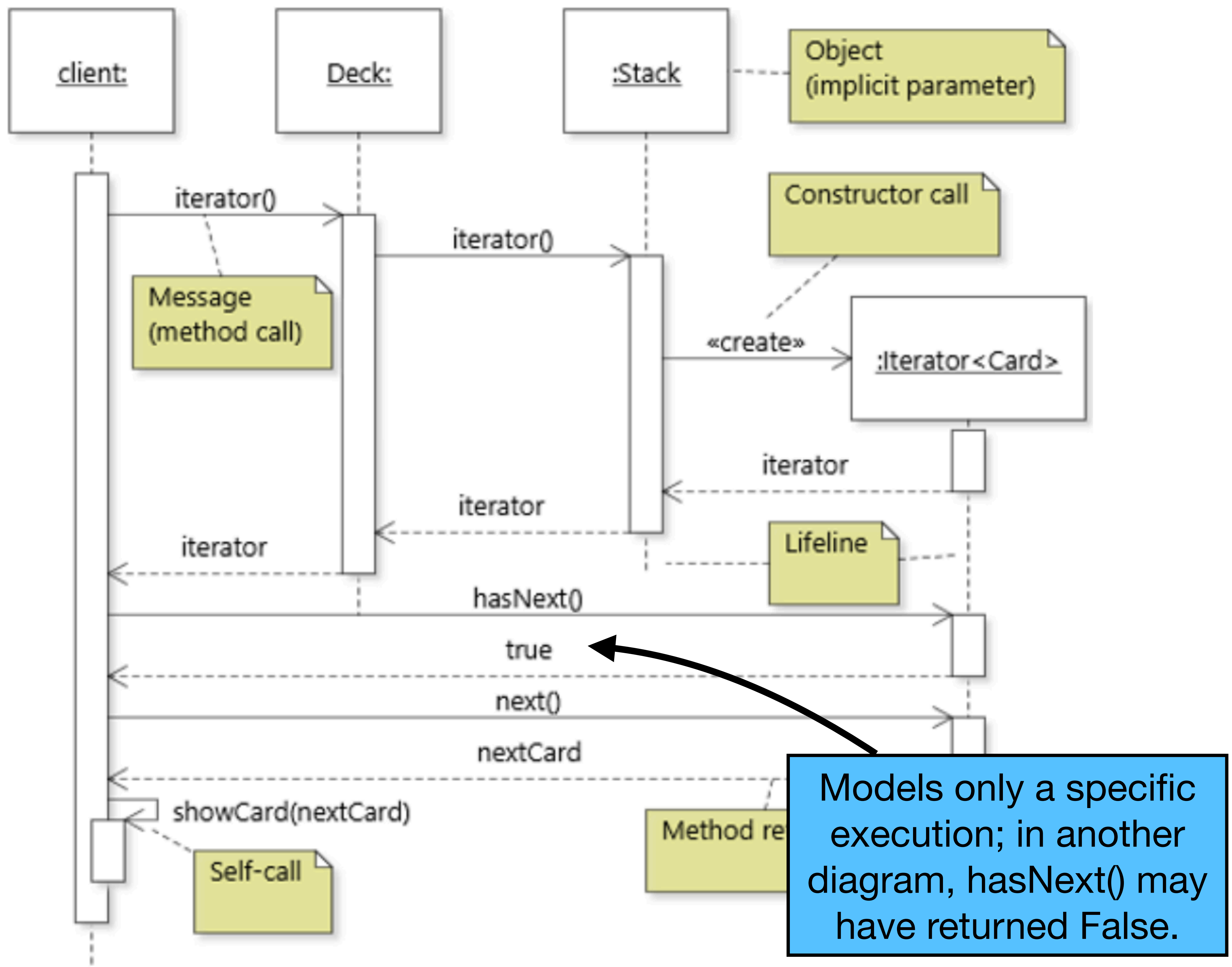
Iterator sequence diagram



Iterator sequence diagram



Iterator sequence diagram



DECORATOR pattern

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.)

Adding functionality to a class

- Problem: What if we have several different kinds of functionality, and we want to add some combination of them to instances of certain class(es)?
  - We'd have to implement each different combination as its own class (either child class or implementing class). That's a lot of work!
- We want to find a nicer, more dynamic way to do this.
  - We want to even support adding and removing functionality **at runtime** (e.g., a user could turn options on or off during gameplay).

Solution #1: Multi-mode class

- One way to do this is to combine all functionalities in a single class (e.g., MultiModeDeck), and just use flags to decide which functionalities to use at any given point.

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) { ... }
    ...
  }
}
```

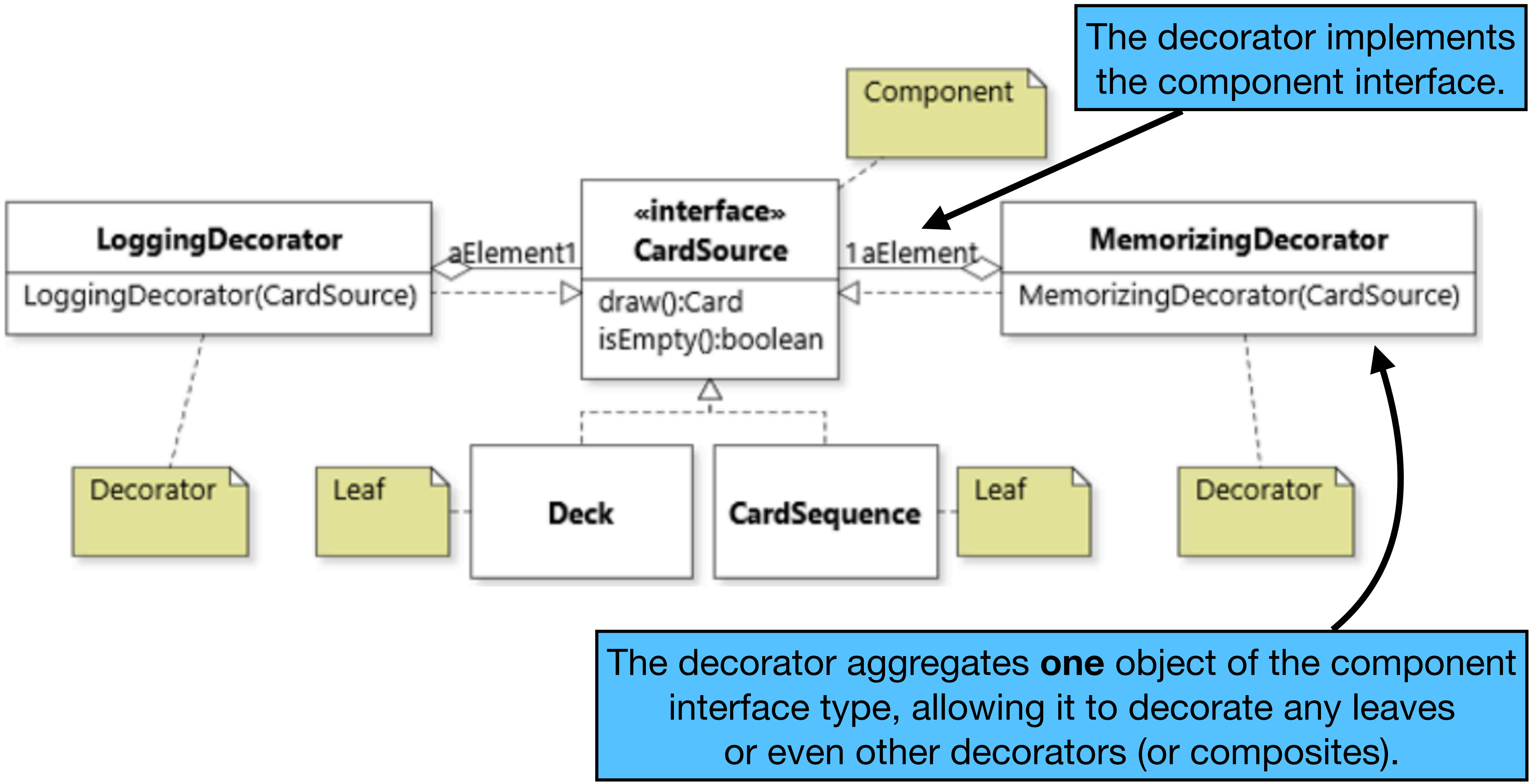
Multi-mode class

- This solution lets us toggle features on and off **at runtime** making the design flexible. But:
  - the state space for objects becomes very complex
    (our state diagrams would be very large!)
  - it violates the principle of separation of concerns.
  - in extreme cases, it could turn a simple class into a God Class (anti-pattern) or could lead to a big Switch Statement (anti-pattern).

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 vs Composite

- Consider the Building/Door example from last class.
- Both Building and Door are MapObjects, and a Building aggregates a single Door.
  - But it could aggregate more things: more doors, or other kinds of MapObjects.
  - It delegates to whichever is most appropriate depending on the user's movement.
- The Building does not modify the door's behaviour; its primary purpose is larger than a single door.

DECORATOR pattern

- In the decorator class, the implementations of the interface methods will delegate the call to the same method on the aggregated object, and then implement their special functionality.
- Each decorator class will add a different functionality.
  - E.g., LoggingDecorator, MemorizingDecorator, ShufflingDecorator.

DECORATOR pattern

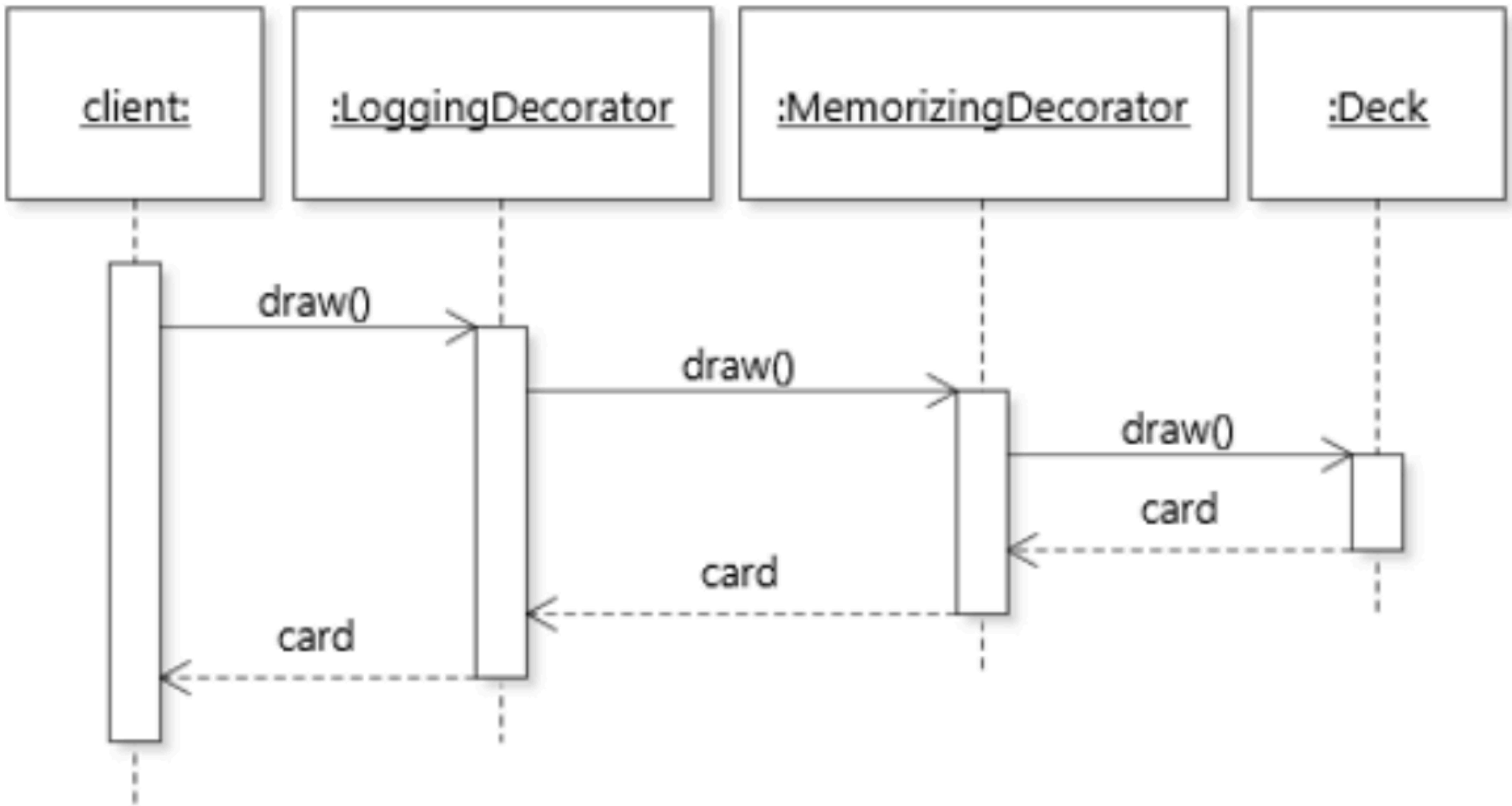
Important to set the component field (aElement) to final, because we don't want to suddenly start decorating a different object.

```
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;
  }
}
```

Combining decorators

- We can easily combine decorations, by having a decorator aggregate another decorator as its component object (i.e., decorate another decorator).
    - Decorations must be independent and strictly additive (and not remove any functionality), otherwise this wouldn't work.

Combining decorators



Decorators and identity

- A decorator object aggregates its undecorated object; it is not the same object, so we should be aware of this when using the == operator if we don't override it.

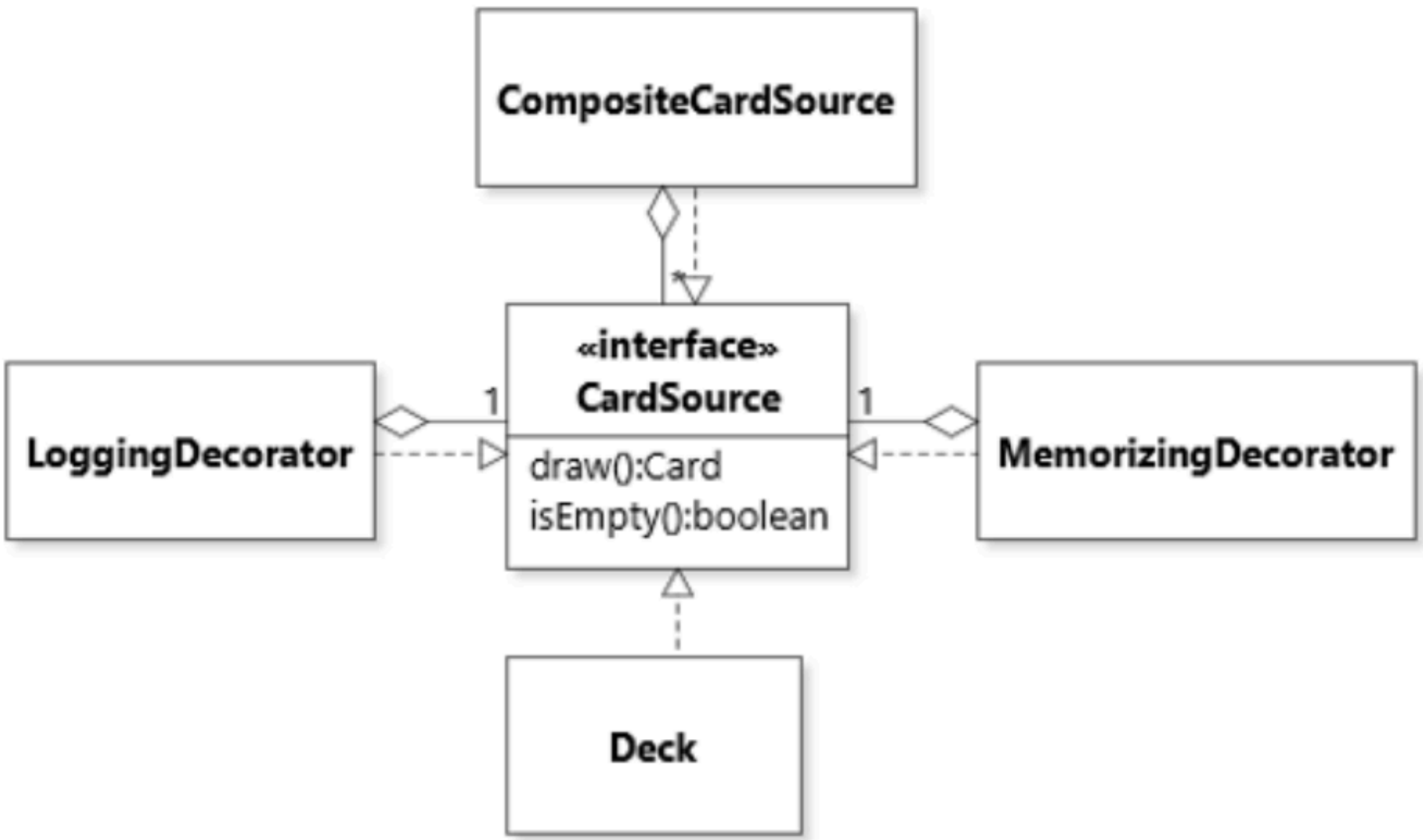
Another example

- Coffee!

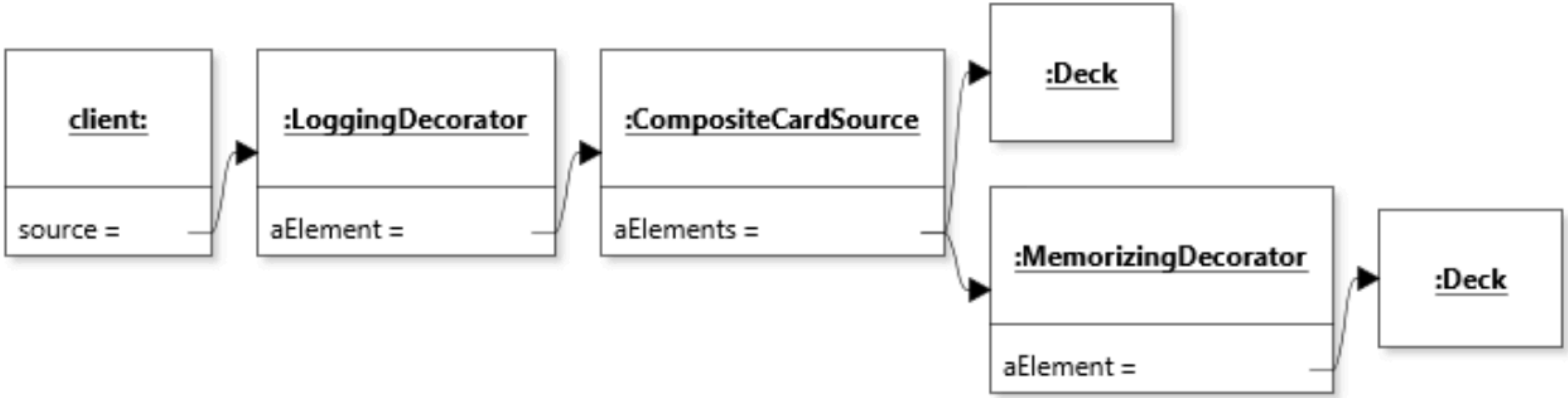
Composite + Decorator

- We can combine the Composite and Decorator patterns if we like, as long as the composite and decorator classes implement the same component interface.

Composite + Decorator

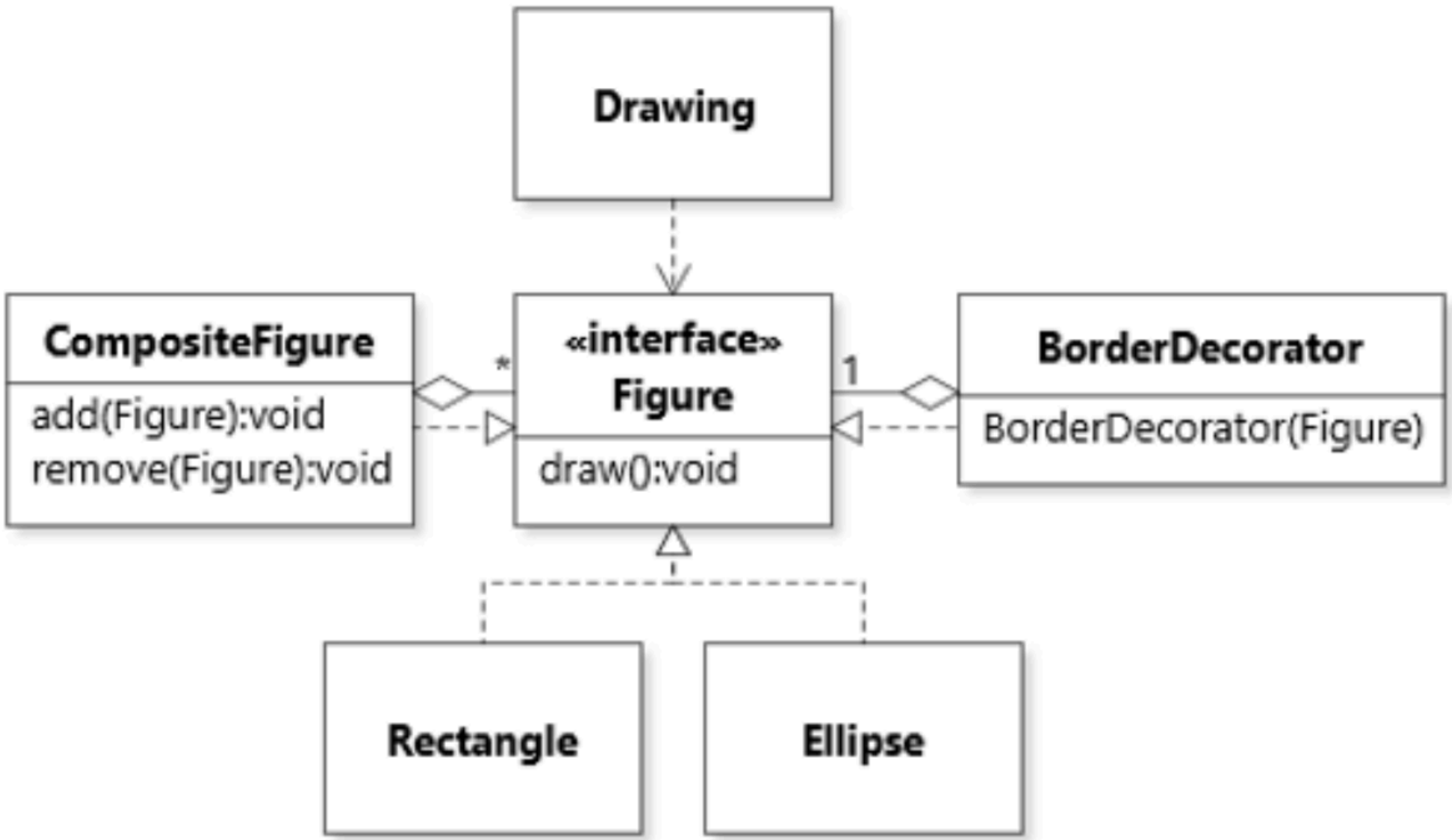


Composite + Decorator



Composite + Decorator

The classic scenario: a picture drawing app



Composite + Decorator

```
public void draw() { // for composite
  for (Figure figure : aFigures) {
    figure.draw();
  }
}

public void draw() { // for decorator
  aFigure.draw();
  // Additional code to draw the border
}
```

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.

References

- Robillard ch. 6.3-6.6 (p. 137-147)
  - Exercises #2-12: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter6.md

Coming up

- Next lecture:
    - More about composition