McGill

COMP 303

Lecture 8

Composition

Winter 2025

slides by Jonathan Campbell

Announcements

- Project server: bugs fixed
  - Pull changes
- Office hours later today
- Brainstorming meetings

Plan for today

- Project
  - Class diagram
  - Persisting state on server
- Composition

Recap

Optional<T>

```
public class Card {
   private Optional<Rank> aRank;
   private Optional<Suit> aSuit;
   public Card() {
      aRank = Optional.empty();
      aSuit = Optional.empty();
         // or Optional.of(value)
         // or Optional.ofNullable(value)
   }
   public boolean isJoker() {
      return !aRank.isPresent();
   }
}
```

To get the value stored within, we call get().

NULL OBJECT pattern

isNull is implemented in all CardSource objects,
but only in NullCardSource will the method return true.

FLYWEIGHT design pattern

- Three components:
  - private constructor, so that clients cannot construct their own objects.
  - static flyweight store (collection of flyweight objects).
  - static access method that returns the appropriate flyweight object. It checks if the object exists yet; if not, it creates it.

SINGLETON pattern

- Three components:
  - Private constructor, so that clients cannot create multiple objects.
  - Global variable that holds the reference to the single class instance.
  - Accessor method, to retrieve the singleton instance.

Flyweight in Python

```
class Card:
    _instances = {} # flyweight store

    def __new__(cls, rank: Rank, suit: Suit):
        self = cls._instances.get((rank, suit))
        if self is None:
            self = cls._instances[(rank, suit)] =
                            object.__new__(Card)
            # initialization code
            self.rank = rank
            self.suit = suit
        return self
```

Singleton in Python

```
class GameModel:
    instance = None

def __new__(cls):
  if cls._instance is None:
    cls._instance = super(GameModel, cls).__new__(cls)
    # Initialization code here.
  return cls._instance
```

Composition

Composition

- **Composite pattern & sequence diagrams (today)**
- Adding functionality to a class (Decorator pattern)
- Composite + Decorator
- Polymorphic copying & the Prototype pattern
- Command pattern & the Law of Demeter

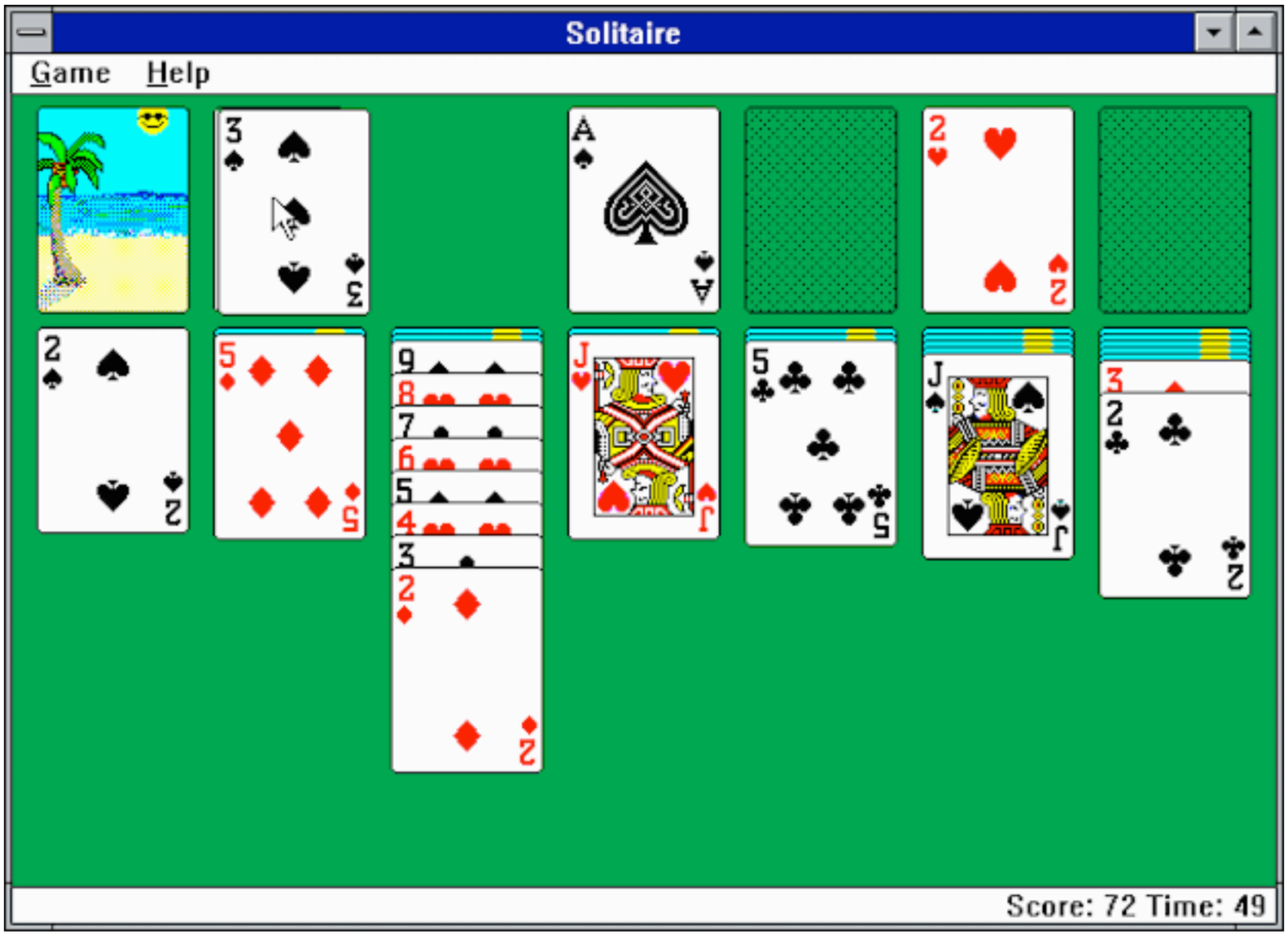
Composition

- 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

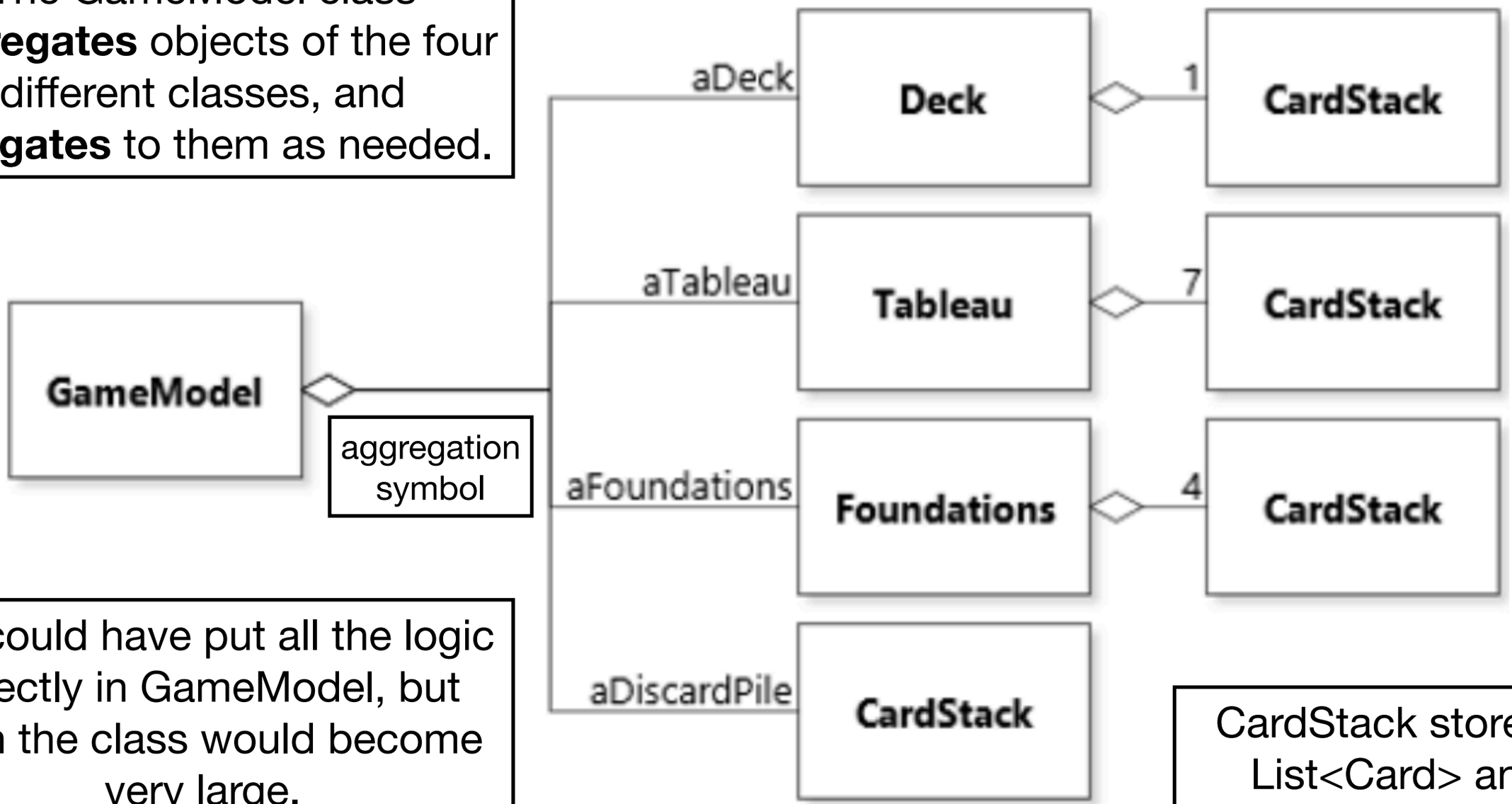


Windows 3.0 solitaire (https://bgr.com/wp-content/uploads/2015/08/windows-solitaire-30.png)

Example: Solitaire

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

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



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.)

Delegation

- GameModel will delegate when needed to its components:

```
public boolean isVisibleInTableau(Card pCard) {
   return aTableau.contains(pCard)
             && aTableau.isVisible(pCard);
}
```

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.

Aggregating CardSources

- We saw that we could write multiple implementing classes for CardSource, like Deck, AggregatedDeck, CardSequence.
  - Or things like FourAces, FaceCards, DeckAndFourAces, ...
- In other words, the set of possible implementations is defined **statically** instead of at run-time.

Aggregating CardSources

- There are a few problems with the design in this case that arise based on the static structure:

- There could be a large number of implementing classes. (DeckAndFourAces is one example of a particular set.)

- Each implementing class requires a class definition, even if used very rarely, leading to unnecessary cluttered code.

- If we wanted a new card source, we'd have to write a new implementing class; we couldn't do anything at run-time.

Aggregating CardSources

- We want to support an open-ended number of configurations.
- To do so, instead of defining classes, we will rely on object composition.
  - We will write a class that is a composition of multiple CardSources, called CompositeCardSource.

CompositeCardSource

- CompositeCardSource will have field(s) to store multiple CardSource objects (e.g., a Deck, a FourAces, etc.).
- CompositeCardSource will also implement the CardSource interface itself.
  - That way, anywhere where a CardSource is expected, the CompositeCardSource can be used.

CompositeCardSource

```
public boolean isEmpty() {
   for (CardSource source : aElements) {
      if (!source.isEmpty()) {
         return false;
      }
   }
   return true;
}
```

CompositeCardSource

```
public Card draw() {
   assert !isEmpty();
   for (CardSource source : aElements) {
      if (!source.isEmpty()) {
         return source.draw();
      }
   }
   assert false;
   return null;
}
```

based on the interface method's preconditions

Adding a card source

```
public CompositeCardSource implements CardSource {
  private final List<CardSource> aElements;
  public CompositeCardSource(List<CardSource> pCardSources) {
    aElements = new ArrayList<>(pCardSources);
  }
}
```

Adding a card source

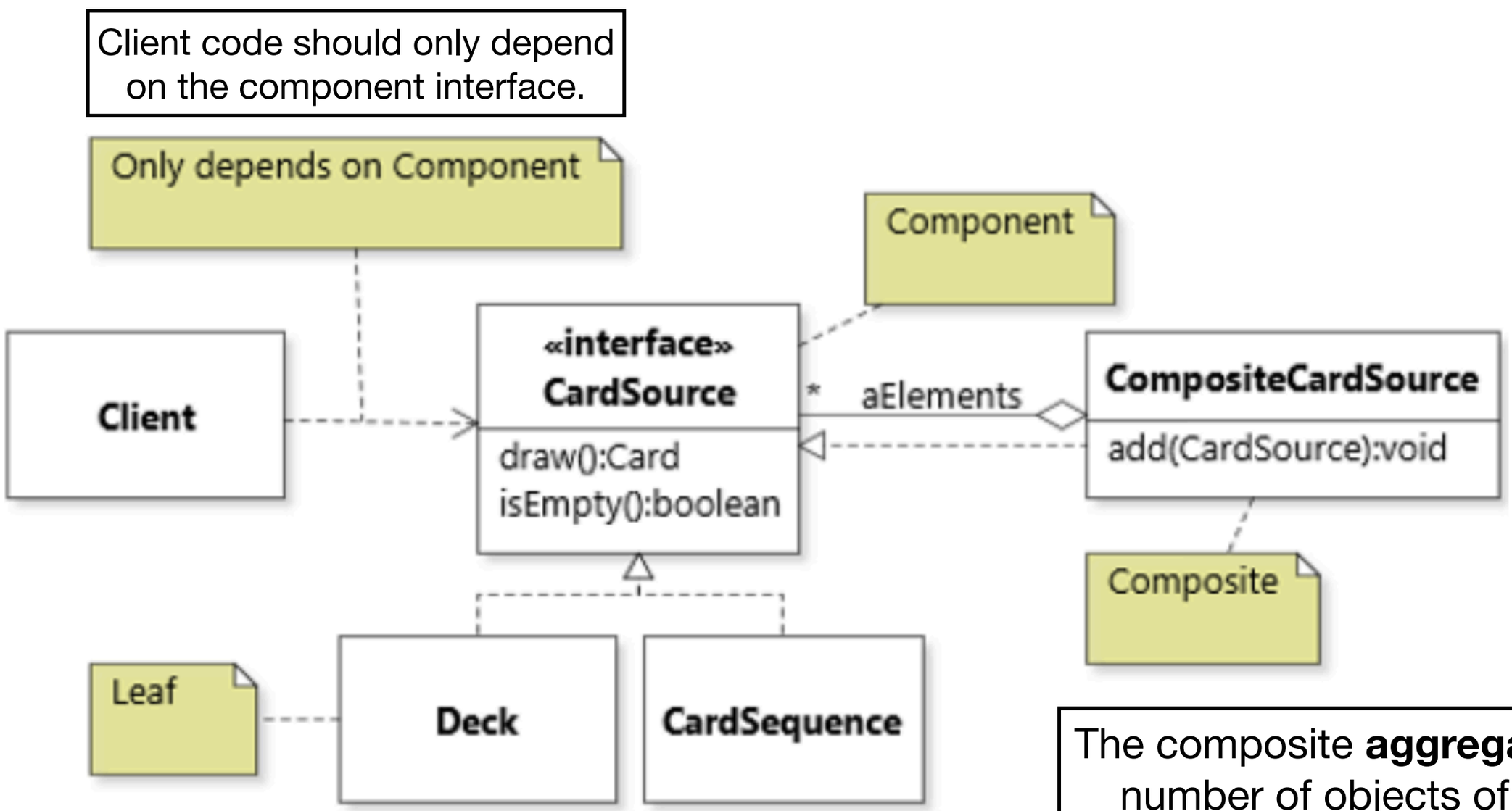
```
public CompositeCardSource implements CardSource {
  // initialize in constructor
  private final List<CardSource> aElements;

  public void add(CardSource pCardSource) {
    aElements.add(pCardSource);
  }
}
```

Can modify at run-time, but makes the design more complicated.

COMPOSITE pattern

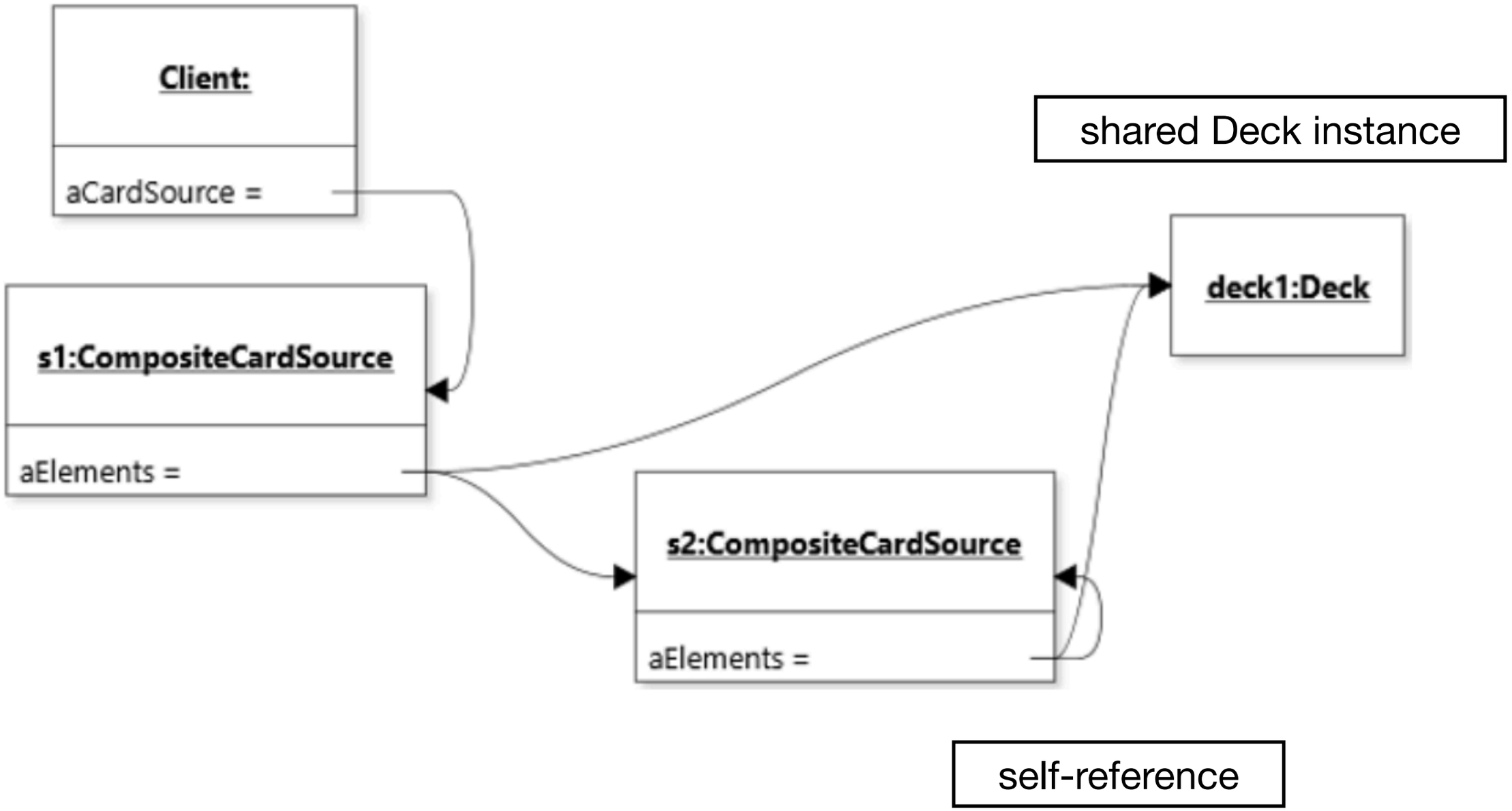
Client code should only depend on the component interface.



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.

Potential pitfalls



Other Composite examples

- In our project code, a Map object aggregates multiple MapObjects, and delegates to them when the player moves into or interacts with them.

- A Building and a Door are both MapObjects, so they can be aggregated by a Map, displayed on the grid and interacted with.

- A Building can itself also contain a Door, and delegate to it when needed.

- Building is a composite and Door is a leaf; both implement the component interface (MapObject).

Other Composite examples

- A SingerSongwriter class may have a sing method that takes a Song object (and calls perform on it).

- There can be different kinds of Songs, including mashup songs which aggregate multiple Songs together!

- LyricalSong, Instrumental, etc., are leaves; MashupSong is a composite, and all implement the component interface (Song).

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.

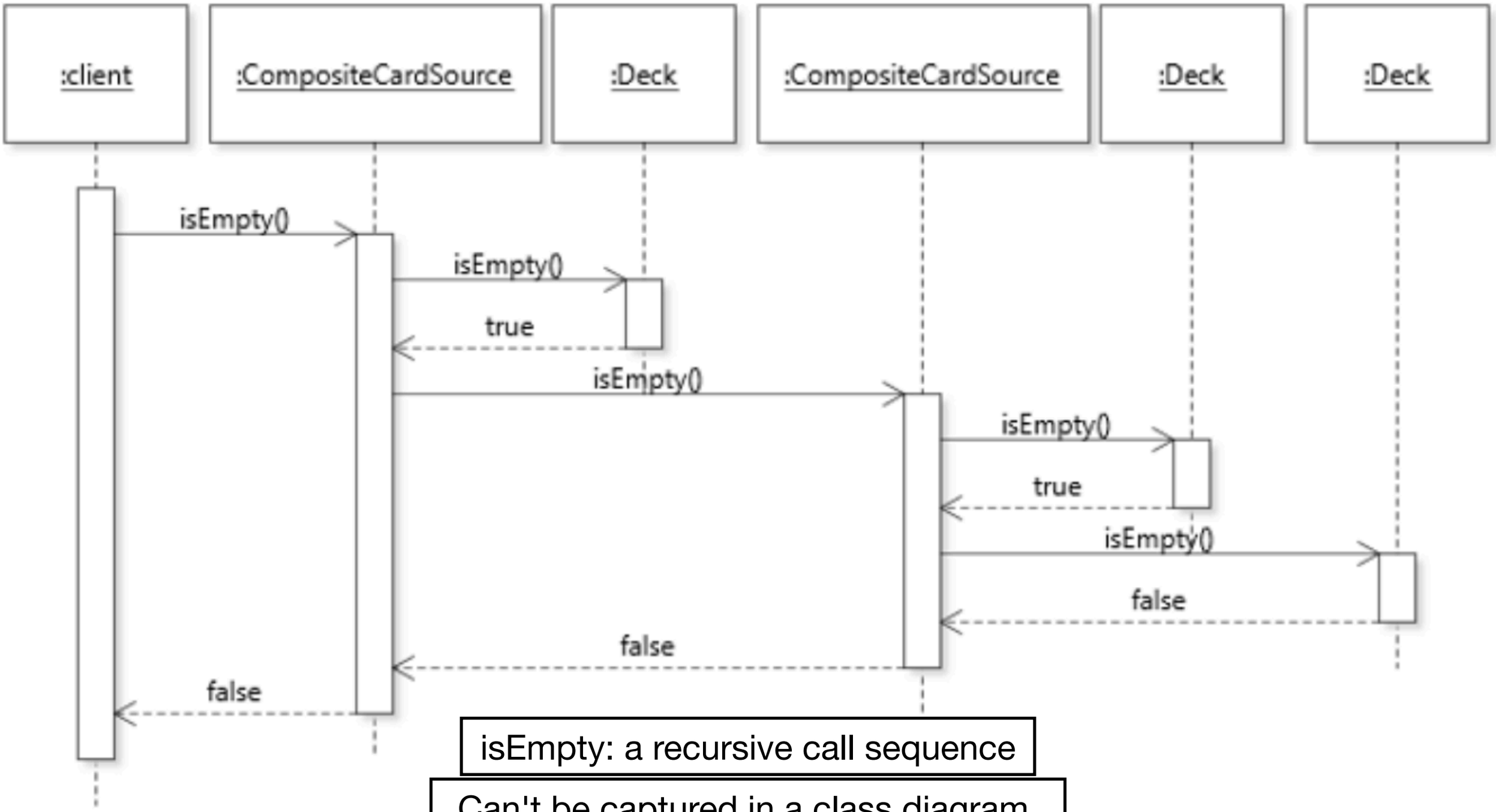
Sequence diagrams

- 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

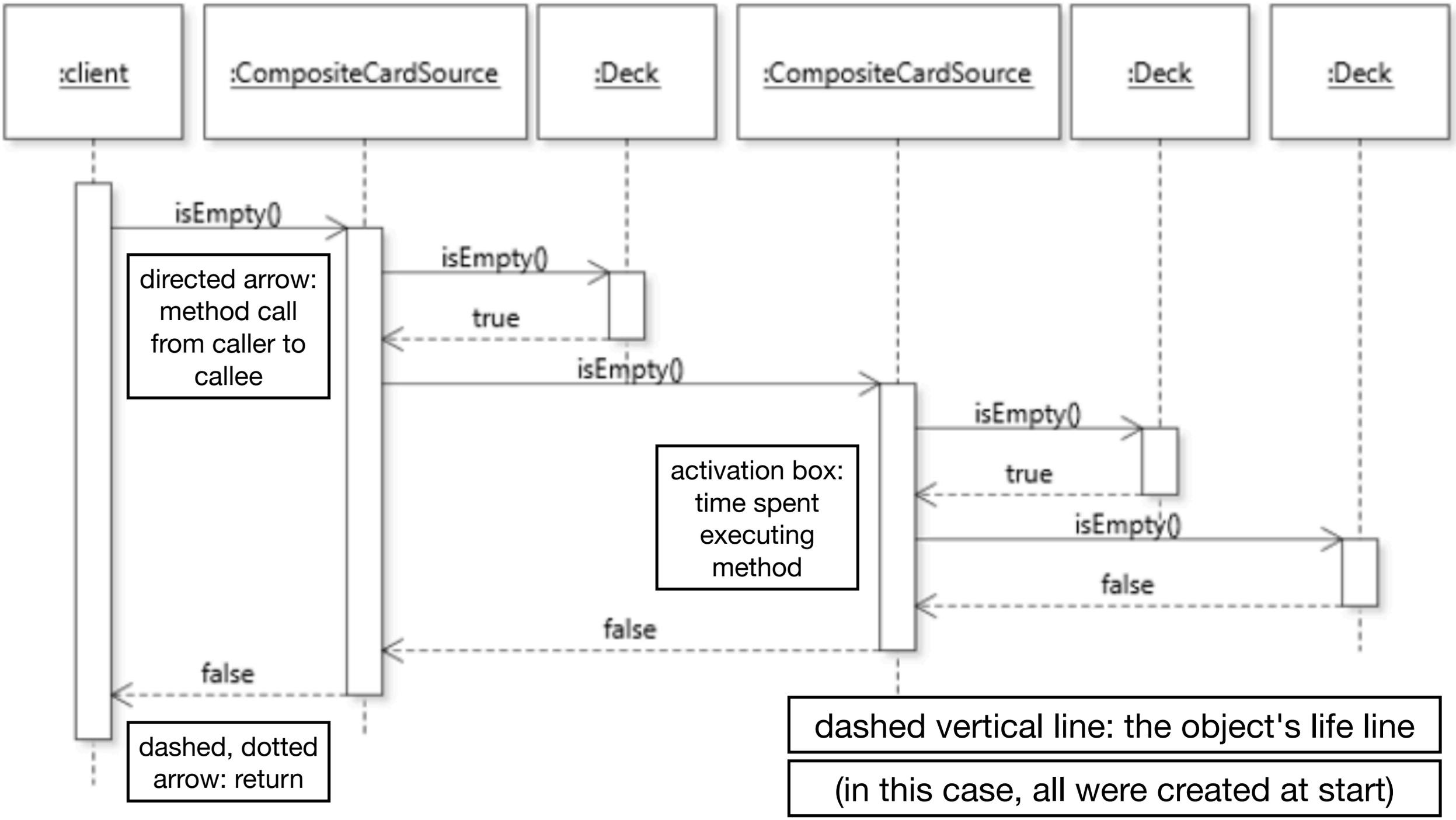
Objects listed at the top, with most informative type names.



isEmpty: a recursive call sequence

Can't be captured in a class diagram.

isEmpty sequence diagram



References

- Robillard ch. 6-6.3 (p. 125-137)
  - Exercise #1: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter6.md

Coming up

- Next lecture:
  - More about composition