COMP 303

Lecture 5

Types & polymorphism

Winter 2025

slides by Jonathan Campbell, adapted in part from those of Prof. Jin Guo

Announcements

- Teams: deadline today
  - If you made an Ed post and found a partner, please resolve the post.
  - If you would like to be matched with partners, please make a private post.
- Proposal instructions coming out Monday.

Plan for today

- Recap from last time
- Types & polymorphism III
  - Iterators in Python
  - Factory methods
  - ISP
- Some exercises
- Project mini-example: Trivia House

Recap

Collections.sort?

```
public class Deck {
   private List<Card> aCards = new ArrayList<>();
   public void sort() {
      Collections.sort(aCards); // ?
   }
}
```

Comparable<T>

```
public interface Comparable<T>
{
    /*
    * Returns a negative integer, zero, or a
    * positive integer if this object is less
    * than, equal to, or greater than the
    * specified object, respectively.
    */
    int compareTo(T o);
}
```

Interfaces

- In general, interfaces should capture the **smallest cohesive slice** of behaviour that is expected to be used by client code.
  - Cohesive: a set of operations that are logically and conceptually related, which will all be used by implementing classes.

- This principle is tied to **separation of concerns**: the idea that each part of a system should handle a single responsibility.
  - One of the advantages of encapsulation.

Comparing cards

```
public class Card implements Comparable<Card> {
   public int compareTo(Card pCard) {
      // compare the cards
   }
}
```

Comparing cards

```
public class ByRankComparator implements Comparator<Card>
{
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
}
```

Comparing cards

```
public class BySuitComparator implements Comparator<Card>
{
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getSuit().compareTo(pCard2.getSuit());
  }
}
```

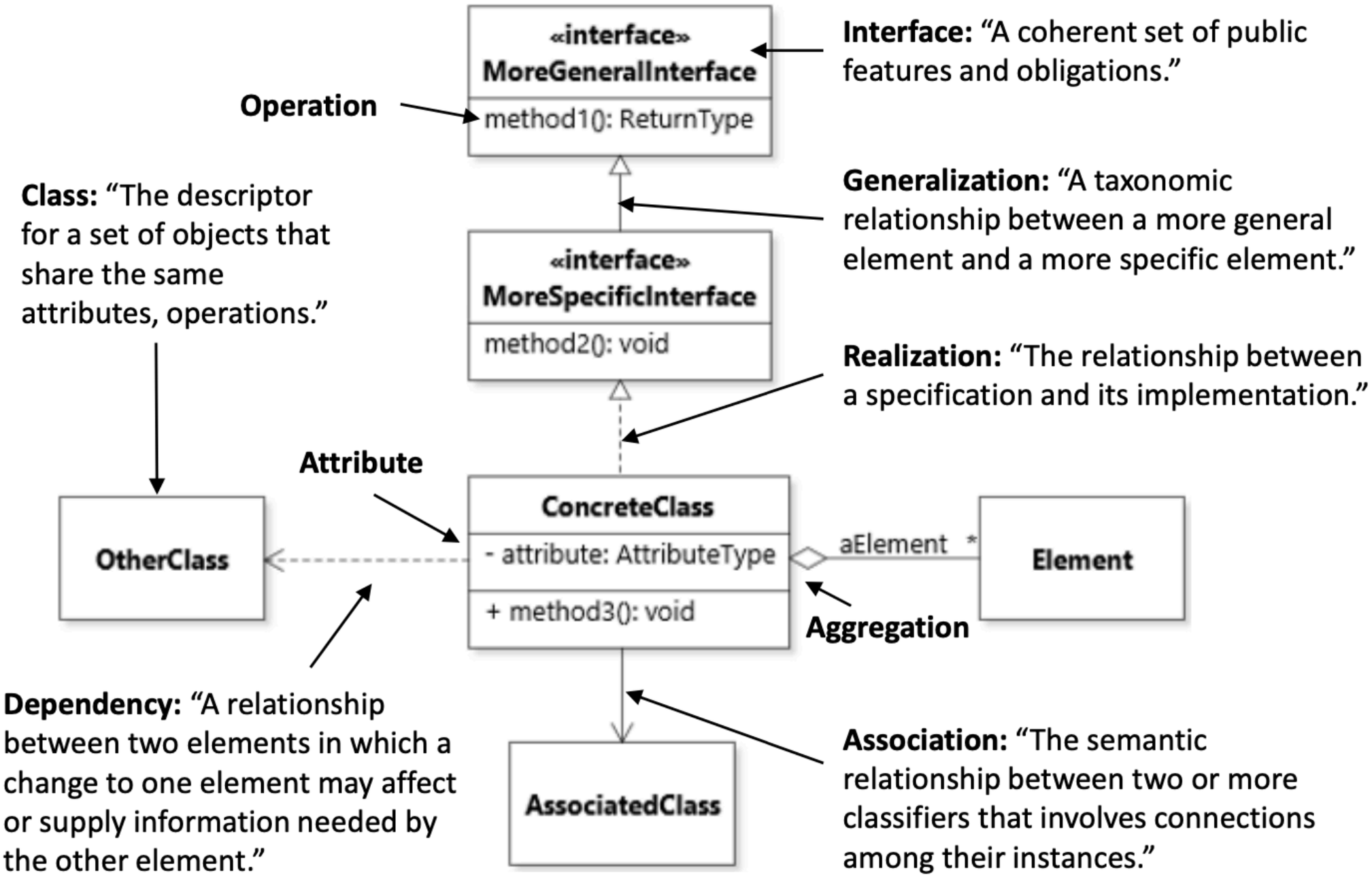
Comparing cards

```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  private Comparator<Card> aComparator;
  public Deck(Comparator<Card> pComparator) {
    aComparator = pComparator;
    shuffle();
  }
  public void sort() {
    Collections.sort(aCards, aComparator);
  }
}
```

STRATEGY design pattern

- If we have a bunch of algorithms to accomplish a task, and want to switch between them flexibly.
  - E.g., switching between different Card comparisons.
  - Or, different AI implementations for a card game bot.
- The main idea is to be able to switch without the client needing to know the algorithm implementation; the algorithms should be interchangeable.

Class diagrams



Comparators as top-level classes

```
public class ByRankComparator implements Comparator<Card>
{
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
}
```

Won't have access to private Card fields.
Have to use getter methods.

Comparators as nested classes

```
public class Card
{
  static class ByRankComparator implements Comparator<Card> {
    public int compare(Card pCard1, Card pCard2) {
      return pCard1.getRank().compareTo(pCard2.getRank());
    }
  }
}

// Collections.sort(aCards, new Card.CompareBySuitFirst());
```

Now has access to private fields.

Comparators as anonymous classes

```
public class Deck {
   public void sort() {
      Collections.sort(aCards, new Comparator<Card>() {
         public int compare(Card pCard1, Card pCard2) {
            /* Comparison code */
         }
      });
   }
}
```

Good if only used in one place.
But, can't store state.

No need for constructor since we are implementing an interface.

Nested class

- A nested class is any class defined inside another (including anonymous classes).

- Why use them?

- If a class is useful to only one other class, then it is logical to embed it into that class.

- It increases encapsulation. By hiding class B inside class A, A's members can be declared private and B can access them; B itself can be hidden from the outside world.

- It places code closer to where it is used.

Iterators

```
public List<Card> getCards() {
   return Collections.unmodifiableList(aCards);
}
```

Even though it returns an unmodifiable version
(so no escaping reference),
it still leaks the internal representation of a Deck.

ITERATOR design pattern

- A mechanism to iterate over the elements stored in an aggregate object, e.g., the cards in a Deck, **without exposing the internal representation** (list, array, etc.).

- It is done by implementing the Iterable<E> interface.

Iterators

- Java classes like the ArrayList already implement Iterator<E>:

```
List<String> stringList = new ArrayList<>();
stringList.add("Apple");
stringList.add("Banana");
stringList.add("Cherry");

Iterator<String> iterator = stringList.iterator();

// Use the iterator to traverse the list
while (iterator.hasNext()) {
    String element = iterator.next();
    System.out.println(element);
}
```

Iterators

- Java classes like the ArrayList already implement Iterator<E>:

```
List<String> stringList = new ArrayList<>();
stringList.add("Apple");
stringList.add("Banana");
stringList.add("Cherry");



for (String string : stringList) {
    System.out.println(string);
}
```

Implementing Iterable<E>

```
public class Deck implements Iterable<Card>
{
    private List<Card> aCards;
    public Iterator<Card> iterator() {
        return aCards.iterator();
    }
}

// later
for (Card card : deck) {
    System.out.println(card);
}
```

Iterator<E>

- We haven't discussed exactly how iteration is done for the built-in types like ArrayList.

- That's because, normally, when we want to iterate, we can just use the standard implementation.

- But if we want to write our own iteration code, we can implement the Iterator<E> interface (not covered here).

Types & polymorphism

Iterators in Python

```
cards : list[Card] = [Card(...), Card(...)]
for card in cards:
    print(card)
```

Iterators in Python

```
class Deck:
    def __init__(self):
        self.cards: list[Card] = []

    def add_card(self, card: Card):
        self.cards.append(card)

    def __iter__(self):
        return iter(self.cards)

if __name__ == '__main__':
    deck = Deck()
    deck.add_card(Card(...))
    deck.add_card(Card(...))
    for card in deck:
        print(card)
```

The Interface Segregation Principle

- Client code should not be forced to depend on interfaces it does not need.

Recall: drawing cards

```
public static List<Card> drawCards(
                        Deck pDeck, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber
                        && pDeck.isEmpty(); i++ ) {
        result.add(pDeck.draw());
    }
    return result;
}
```

Recall: drawing cards

```
public interface CardSource {
  /**
  * Returns a card from the source.
  *
  * @return The next available card.
  * @pre !isEmpty()
  */
  Card draw();
  /**
  * @return True if there is no card in the
  * source.
  */
  boolean isEmpty();
}
```

Recall: drawing cards

```
public static List<Card> drawCards(CardSource
                                   pSource, int pNum) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNum
                        && !pSource.isEmpty(); i++ ) {
        result.add(pSource.draw());
    }
    return result;
}
```

CardSource

```
public interface CardSource {

  Card draw();
  /**
  * @return True if there is no card in the
  * source.
  */
  boolean isEmpty();
}
```

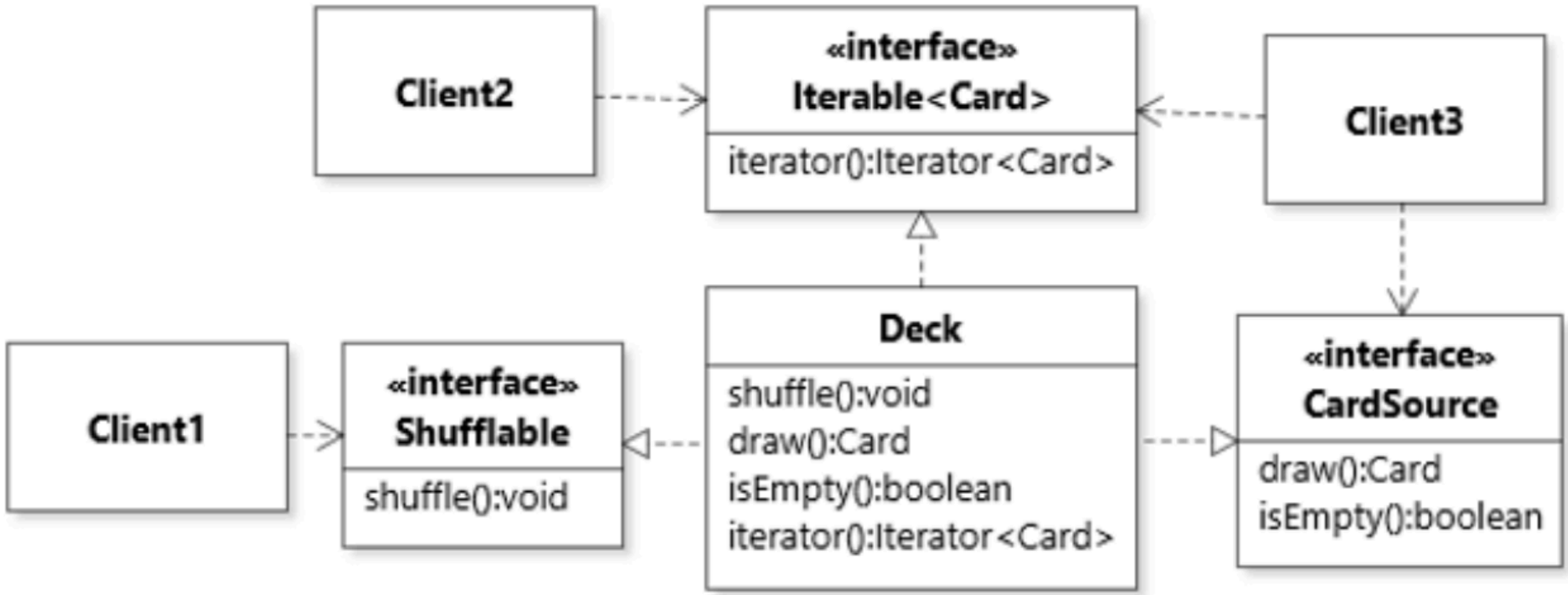
CardSource

```
public interface CardSource {

   void shuffle();
   Card draw();
   /**
   * @return True if there is no card in the
   * source.
   */
   boolean isEmpty();
}
```

What if we want drawCards to draw from a class that does not need shuffling?

Separation of concerns

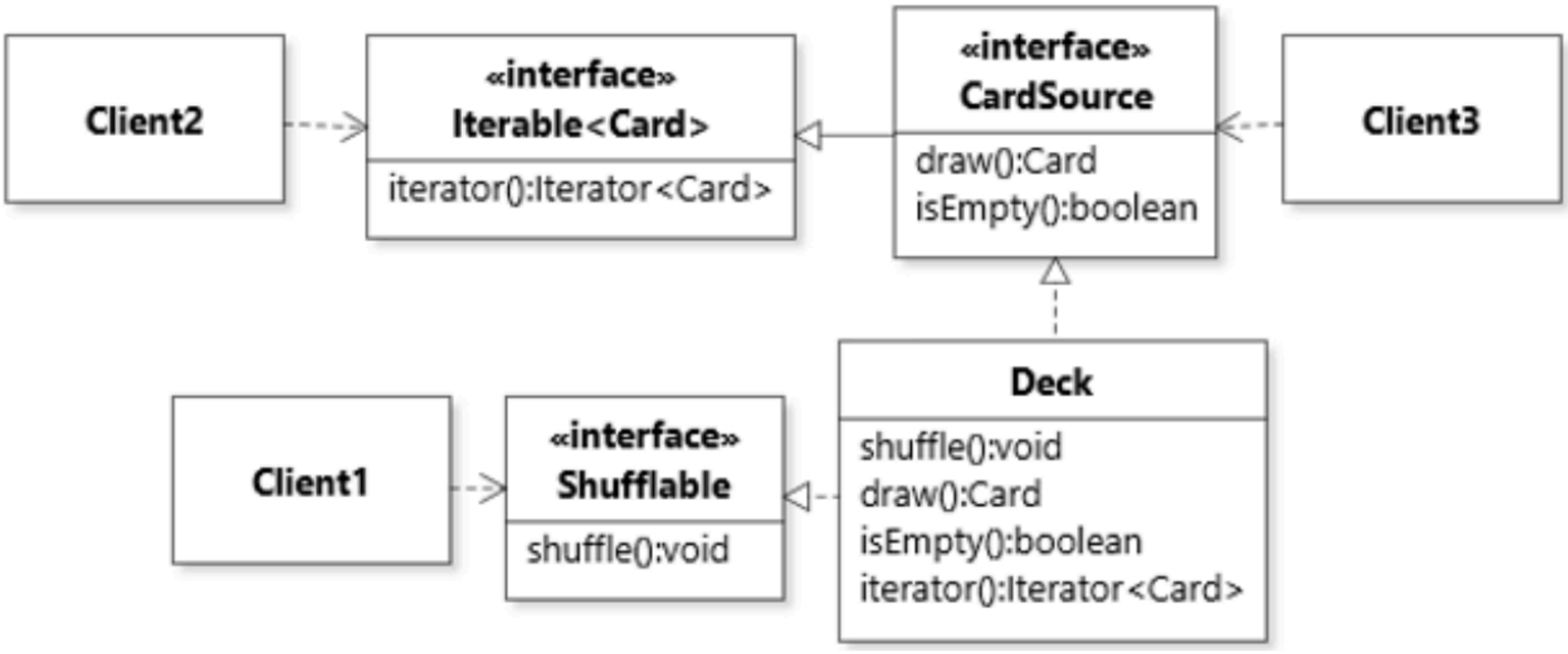


Problem

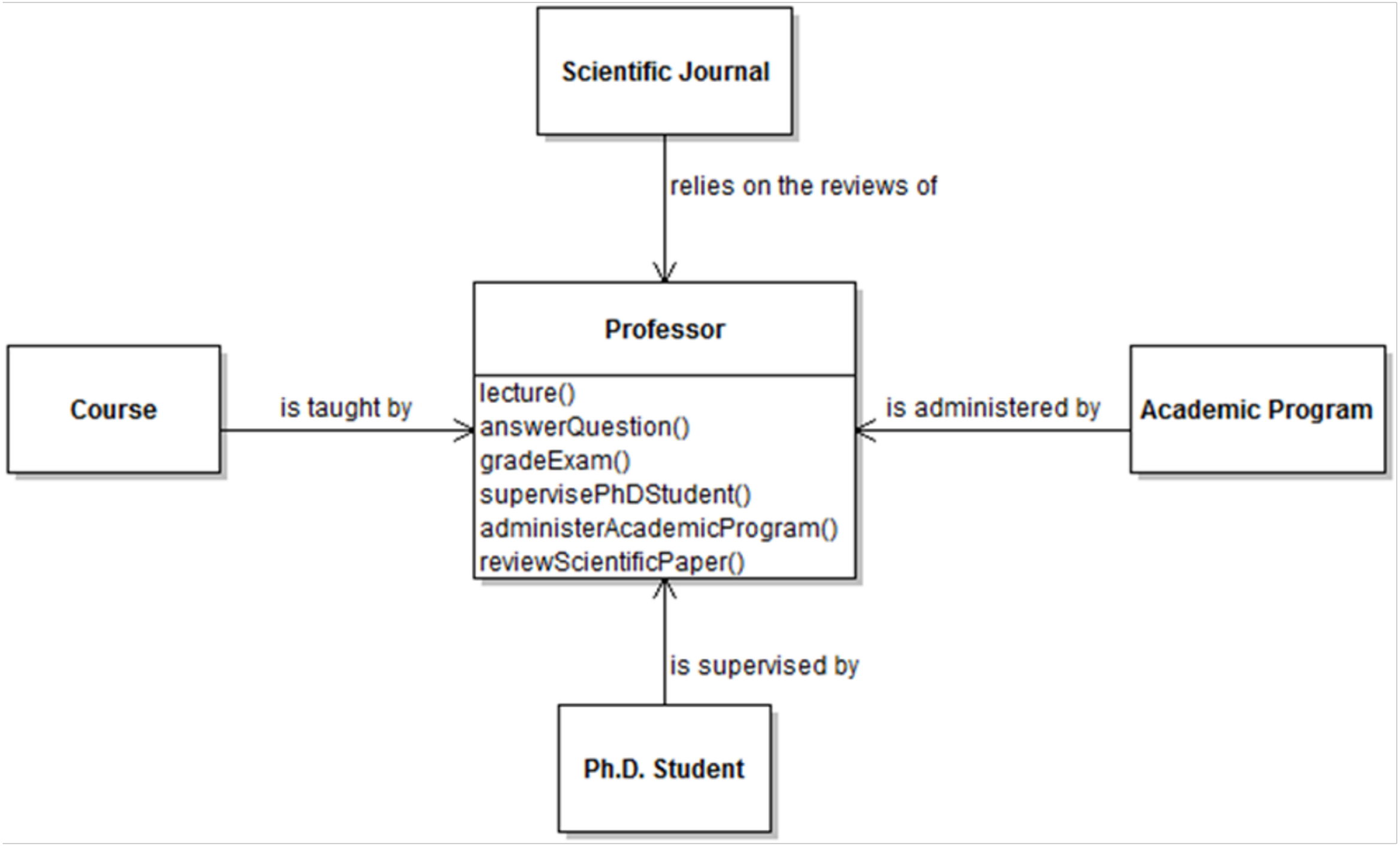
- Client3 needs to iterate and also draw cards from a CardSource. To do so, it would have to do a cast (which could be unsafe depending on the argument):

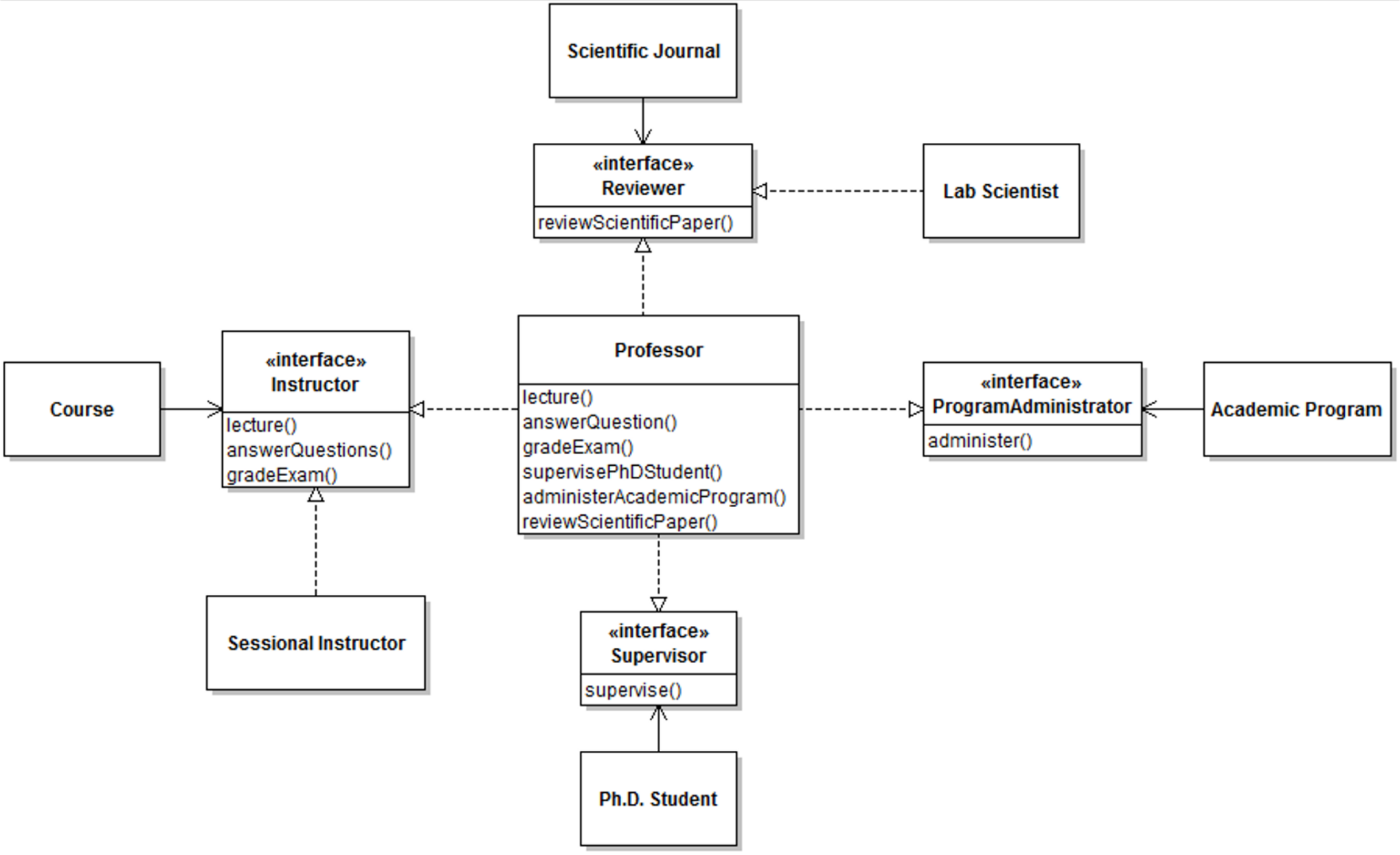
```
public void displayCards(CardSource pSource) {
   if (!pSource.isEmpty()) {
      pSource.draw();
      for (Card card : (Iterable<Card>) pSource) {
         ...
      }
   }
}
```

Solution



If a lot of code that uses CardSource also needs to iterate (but not the other way around), then it makes sense to have CardSource extend Iterable.


Scientific Journal
relies on the reviews of
Professor
lecture()
answerQuestion()
gradeExam()
supervisePhDStudent()
administerAcademicProgram()
reviewScientificPaper()
Course
is taught by
is administered by
Academic Program
is supervised by
Ph.D. Student


Scientific Journal
«interface»
Reviewer
reviewScientificPaper()
Lab Scientist
Course
«interface»
Instructor
lecture()
answerQuestions()
gradeExam()
Professor
lecture()
answerQuestion()
gradeExam()
supervisePhDStudent()
administerAcademicProgram()
reviewScientificPaper()
«interface»
ProgramAdministrator
administer()
Academic Program
Sessional Instructor
«interface»
Supervisor
supervise()
Ph.D. Student

Comparators as nested classes

```
public class Card
{
  static class ByRankComparator implements Comparator<Card> {
    public int compare(Card pCard1, Card pCard2) {
      return pCard1.getRank().compareTo(pCard2.getRank());
    }
  }
}

// Collections.sort(aCards, new Card.CompareBySuitFirst());
```

Now has access to private fields.

Comparators as anonymous classes

```
public class Deck {
   public void sort() {
      Collections.sort(aCards, new Comparator<Card>() {
         public int compare(Card pCard1, Card pCard2) {
            /* Comparison code */
         }
      });
   }
}
```

Good if only used in one place.
But, can't store state.

But: now Card comparison logic is in the Deck class.

Comparators as anonymous classes

```
public class Card {
  public static Comparator<Card> createByRankComparator() {
    return new Comparator<Card>() {
      public int compare(Card pCard1, Card pCard2) {
        /* Comparison code */
      }
    };
  }
}
public class Deck {
  public void sort() {
    Collections.sort(aCards, Card.createByRankComparator());
  }
}
```

Known as a static factory method.

Factory: a method that creates and returns an object.

Exercises

Exercises

https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter3.md

Project mini-example

References

- Robillard ch. 3.9 (p. 62-66)
  - Exercises 1-10: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter3.md

Coming up

- Next lecture:
  - More about types and polymorphism