

CS 417

Design Patterns

Design patterns: Strategy cont., Factory, and Iterator

Dr. Chad Williams
Central Connecticut State University

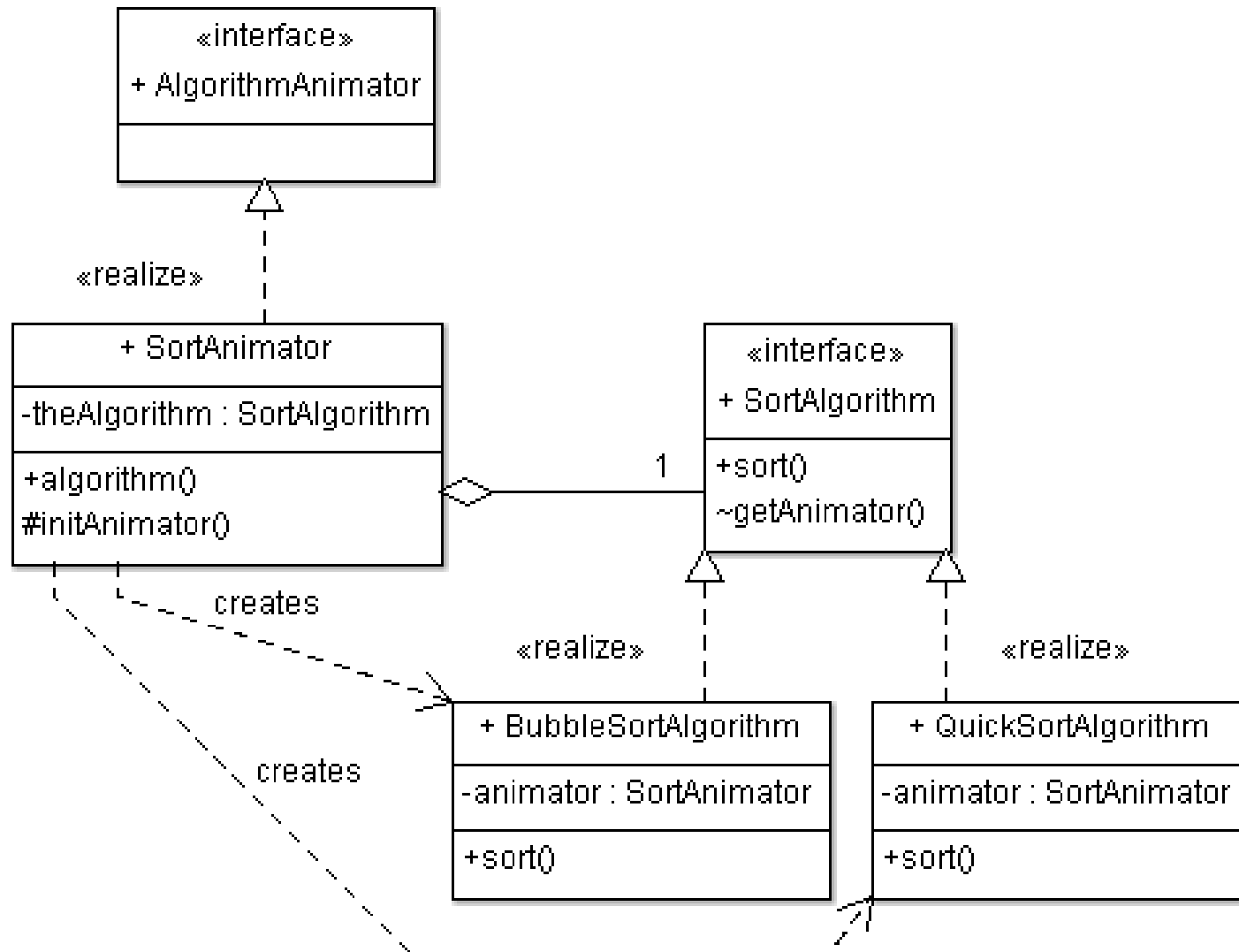
Agenda

- Design pattern: Strategy cont.
- Design pattern: Factory
- Design pattern: Iterator

Strategy example

- Sorting algorithm animation
- Application displays an animation of how the elements within an array change as the algorithm runs
- Should be able to switch algorithms

Encapsulating sorting algorithm



Creating instances of concrete algorithms

```
public class SortAnimator implements
    AlgorithmAnimator{
    protected SortAlgorithm theAlgorithm;
    protected void initAnimator(){
        algName = "BubbleSort";
        String at = getParameter("alg");
        if (at != null){
            algName = at;
        }
        if ("BubbleSort".equals(algName)) {
            theAlgorithm = new BubbleSortAlgorithm(this);
        } else if ("QuickSort".equals(algName)) {
            theAlgorithm = new QuickSortAlgorithm(this);
        } else {
            theAlgorithm = new BubbleSortAlgorithm(this);
        }
    }
}
```

Design analysis

- Algorithms can be switched without impacting animation code
- While majority of code abstracted, tightly coupled in creation of concrete algorithms
 - If new algorithms added, `initAnimator` code must be changed as well to be used
 - Goal be able to add sorting algorithms without changing code in `SortAnimator`

Separating creation

- Better alternative is to separate creation of concrete classes
- Factory pattern separates creation and encapsulates concrete classes from other code
- Decoupled code allows concrete classes to be added or changed with single point of code impact

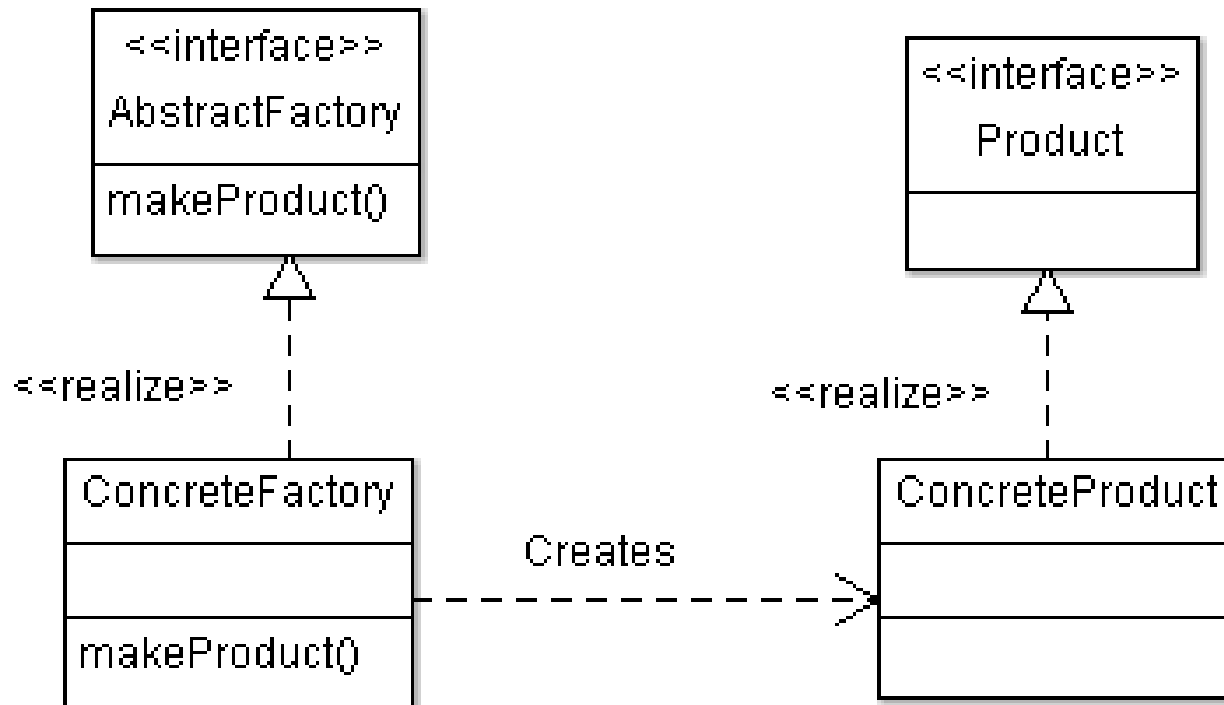
Factory pattern

- **Category:** Creational design pattern
- **Intent:** Define an interface for creating objects but let subclasses decide which class to instantiate and how
- **Applicability:** Should be used when a system should be independent of how its products are created

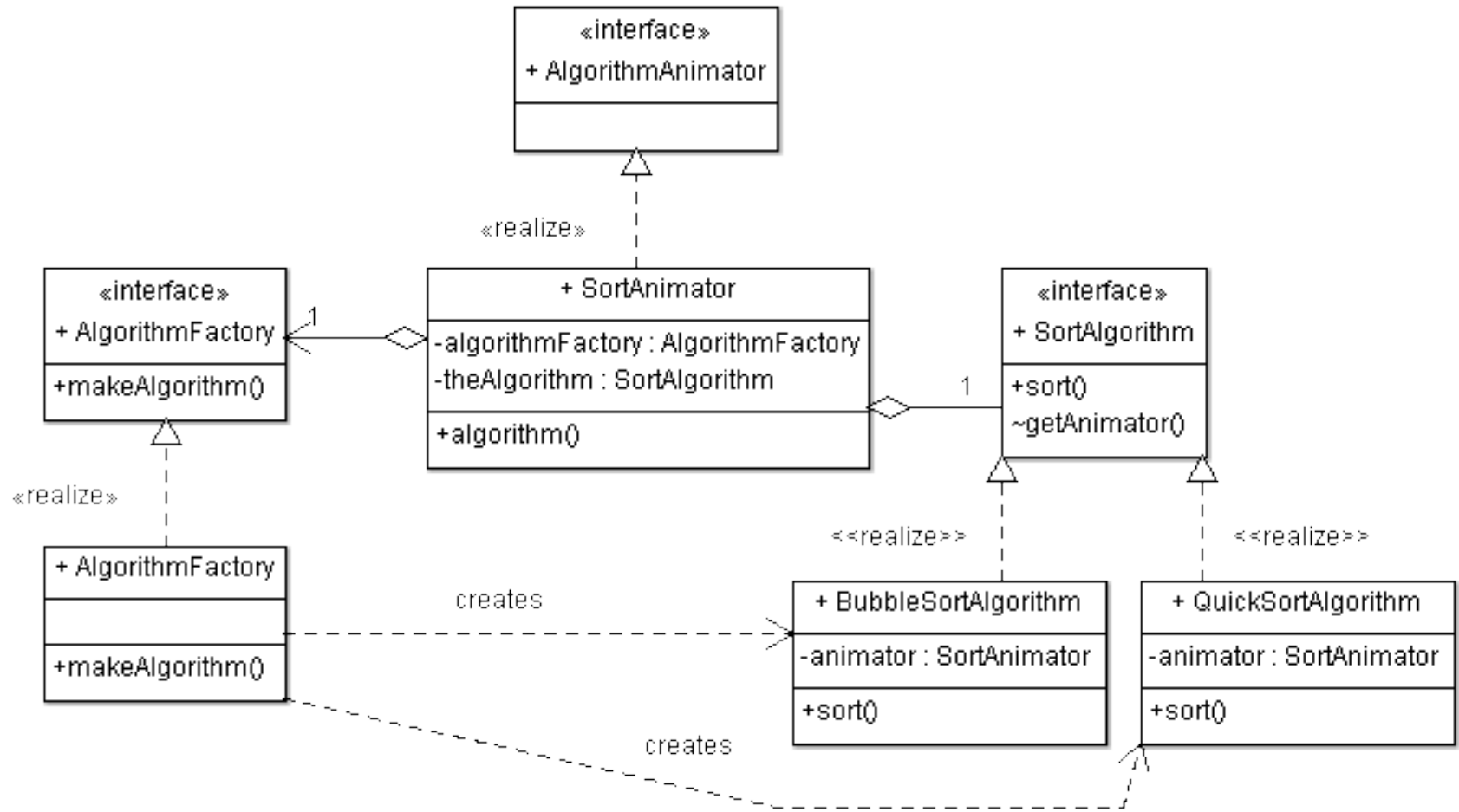
Factory pattern participants

- **Product** – Defines an interface of objects the factory will create
- **ConcreteProduct** – Implements the Product interface
- **AbstractFactory** – Defines a factory method that returns an object of type Product
- **ConcreteFactory** – Overrides the factory method to return an instance of ConcreteProduct

Factory UML



Example UML



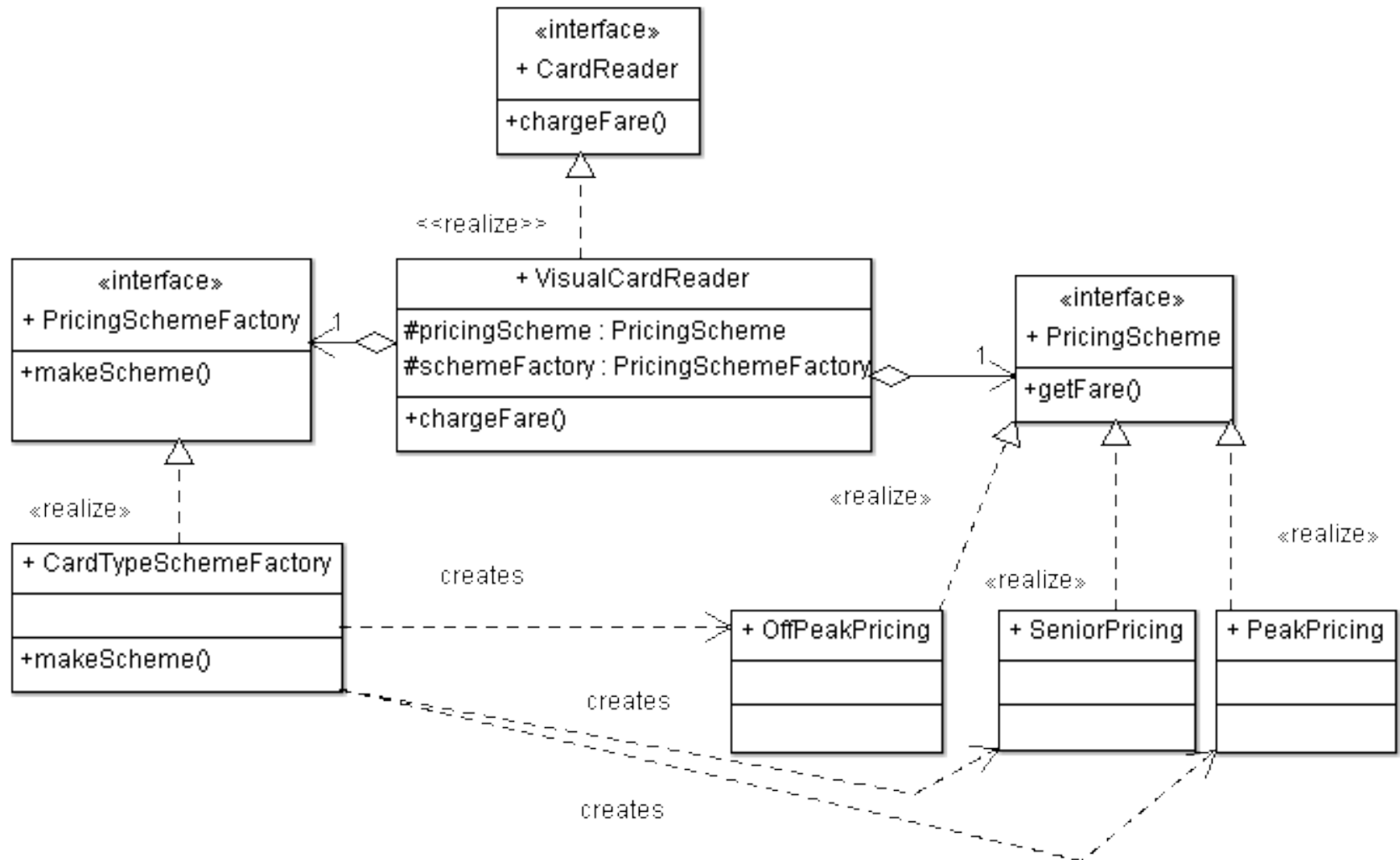
Revised SortAnimator

```
public class SortAnimator implements AlgorithmAnimator{
    private SortAlgorithm theAlgorithm;
    private SortAlgorithmFactory algorithmFactory;
    protected void initAnimator(){
        String at = getParameter("alg");
        algorithmFactory = new SortAlgorithmFactory();
        theAlgorithm = algorithmFactory.makeAlgorithm(at);
    }
}
```

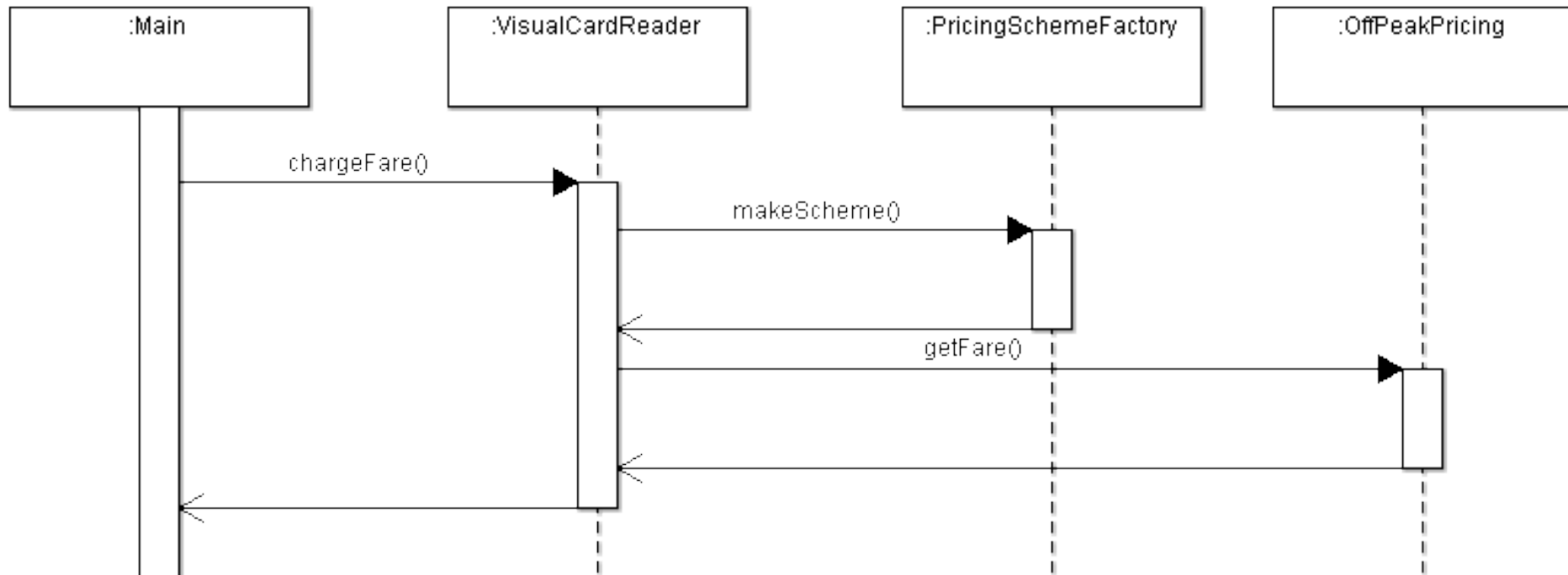
Card reader problem

Chicago is creating a new mass transit fare system. The system requires users to have an fare card that can be **read by multiple types of systems** (such as swiped, visual, etc). For one of these types, **visual**, they want the system to provide visual traits of the card and **determine the fare pricing scheme (Off peak, peak, senior)** that should be used for that card for the specified request. Note they want the **flexibility to have the look of fare cards to change and add additional pricing schemes** in the future. Draw the **class diagram** and sample **sequence** from a class Main.

Card reader solution



Card reader sequence



Group work

For a new ATM that can use **either** a specific **fingerprint** (i.e. each finger specifies a different account) or old fashion enter an **account number**. Once they have provided their **info to specify the account** and an instance of the **Account** of type **Checking** or **Credit** is returned where the type of account is determined based on the account number/fingerprint. Design for future flexibility in account selection method and account types. **Draw the class diagram and a sample sequence assuming the ATM class has chosen fingerprint access.**

Design guideline

- Program to an interface, not an implementation
 - Separate interface from implementation
 - Clients access functionalities via interface not directly on class
 - Implementation hidden from clients
- Programming to a class → context-specific, inflexible solutions
- Programming to an interface → general, extensible, reusable solutions

Enumerating elements

- Scenario – Given a group of Objects
 - Operation - Loop through elements
 - Common operation common to many different data structures
- Multiple implementations possible, used in multiple contexts making it a prime candidate for generalization

Solution 1 - direct access

```
LinkedList list;  
for (LinkedList.Node cur = list.head;  
     cur != null; cur = cur.next) {  
    System.out.println(cur.element);  
}
```

- Solution works for LinkedList but tightly coupled to implementation breaking encapsulation
- Also would have to completely recode if a different data structure was wanted

Solution 2 - Iterate via Method invocation

```
public class IterList extends LinkedList{
    public void reset(){ cur = head; }
    public Object next(){
        Object obj = null;
        if (cur != null){
            obj = cur.element;
            cur = cur.next;
        }
        return obj;
    }
    public boolean hasNext(){
        return (cur != null);
    }
    protected Node cur;
}
```

- Encapsulates implementation but still strongly tied to context of implementation (ex. Looping within a loop)

Solution 3 - Separate iterator from List

```
public class LinkedListIterator{
    public LinkedListIterator(LinkedList list){
        this.list = list;
        cur = list.head;
    }
    public Object next(){
        Object obj = null;
        if (cur != null){
            obj = cur.element;
            cur = cur.next;
        }
        return obj;
    }
    public boolean hasNext(){
        return (cur != null);
    }
    protected LinkedList.Node cur;
    protected LinkedList list;
}
```

- Fixes the multiple iterator problem but iterator is still tightly coupled with LinkedList

Solution 4: Generalization through abstract coupling

```
interface Iterator{  
    Object next();  
    boolean hasNext();  
    void remove();  
}
```

- Now concrete iterators such as a `LinkedListIterator` or a `TreeIterator` can implement the `Iterator` interface and the client doesn't have to change its code despite change in implementation

Solution 4 cont.

```
public interface List{
    public Iterator iterator();
    ...
}
public class LinkedList implements List{
    public Iterator iterator(){
        return new LinkedListIterator();
    }
    private class LinkedListIterator implements
    Iterator{
    }
}
```

- Now any number of structures that support List interface can also return a way to iterate through them without clients being able to access internals at all

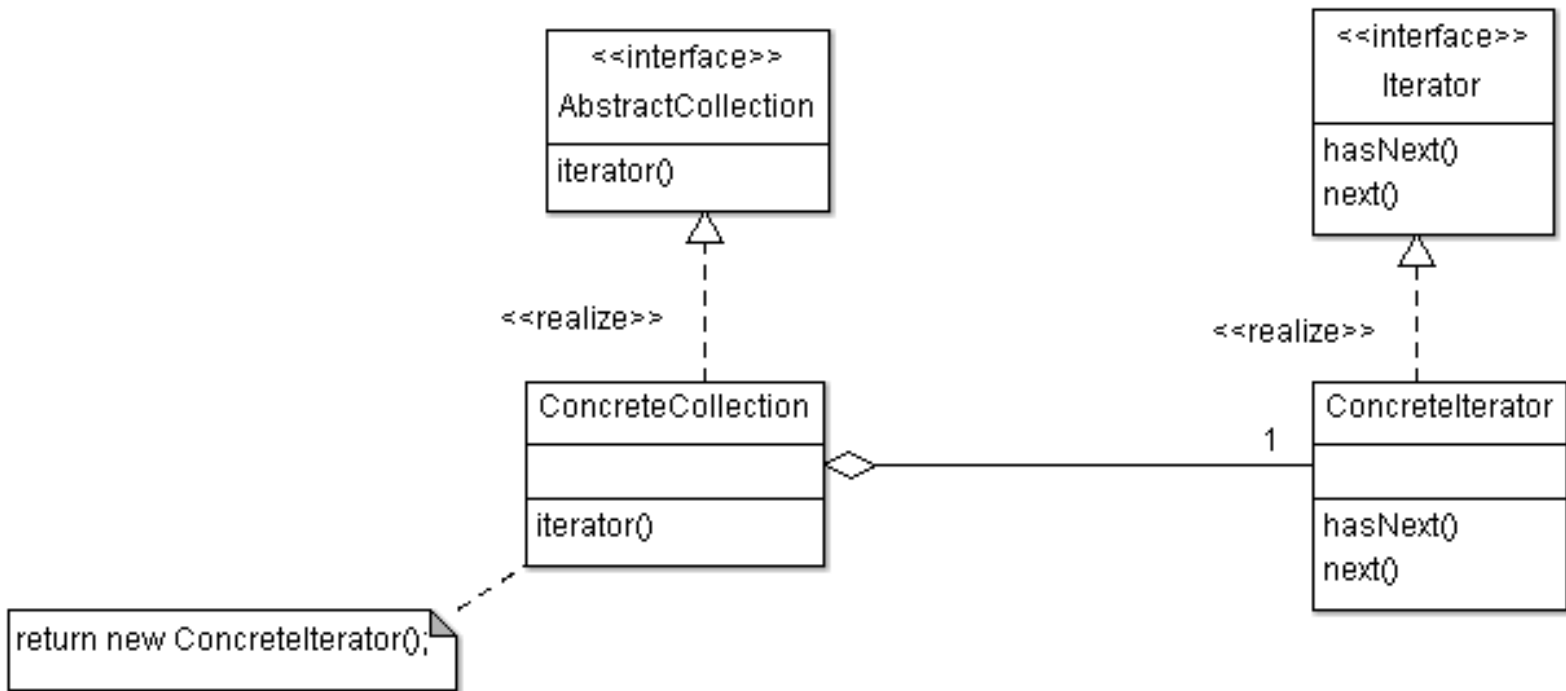
Design pattern: Iterator

- **Category** – Behavioral design pattern
- **Intent** – Provide a way to access the elements of a collection sequentially
- **Applicability** – Should be used if...
 - To access contents of a collection without exposing its internal representation
 - Support multiple traversals of a collection (with Trees preOrder(),postOrder())
 - Provide uniform interface for traversing different collections (polymorphic iteration)

Iterator participant descriptions

- **Iterator** – defines interface for accessing and traversing the elements
- **ConcreteIterator** – implements iterator interface and keeps track of current position in traversal
- **AbstractCollection** – defines interface for creating a concrete iterator
- **ConcreteIterator** – implements the iterator method and returns an instance of the proper ConcreteIterator

Iterator UML



Tree example

- Create UML class diagram extension for an tree data structure support of the iterator pattern.
- Write pseudo code to support iterator pattern for a tree data structure

Group work

- Create UML class diagram extension for an array data structure support of the iterator pattern.
- Write pseudo code to support iterator pattern for an array data structure