Design Patterns

Design patterns: Strategy cont. Abstract Factory pattern

Dr. Chad Williams
Central Connecticut State University

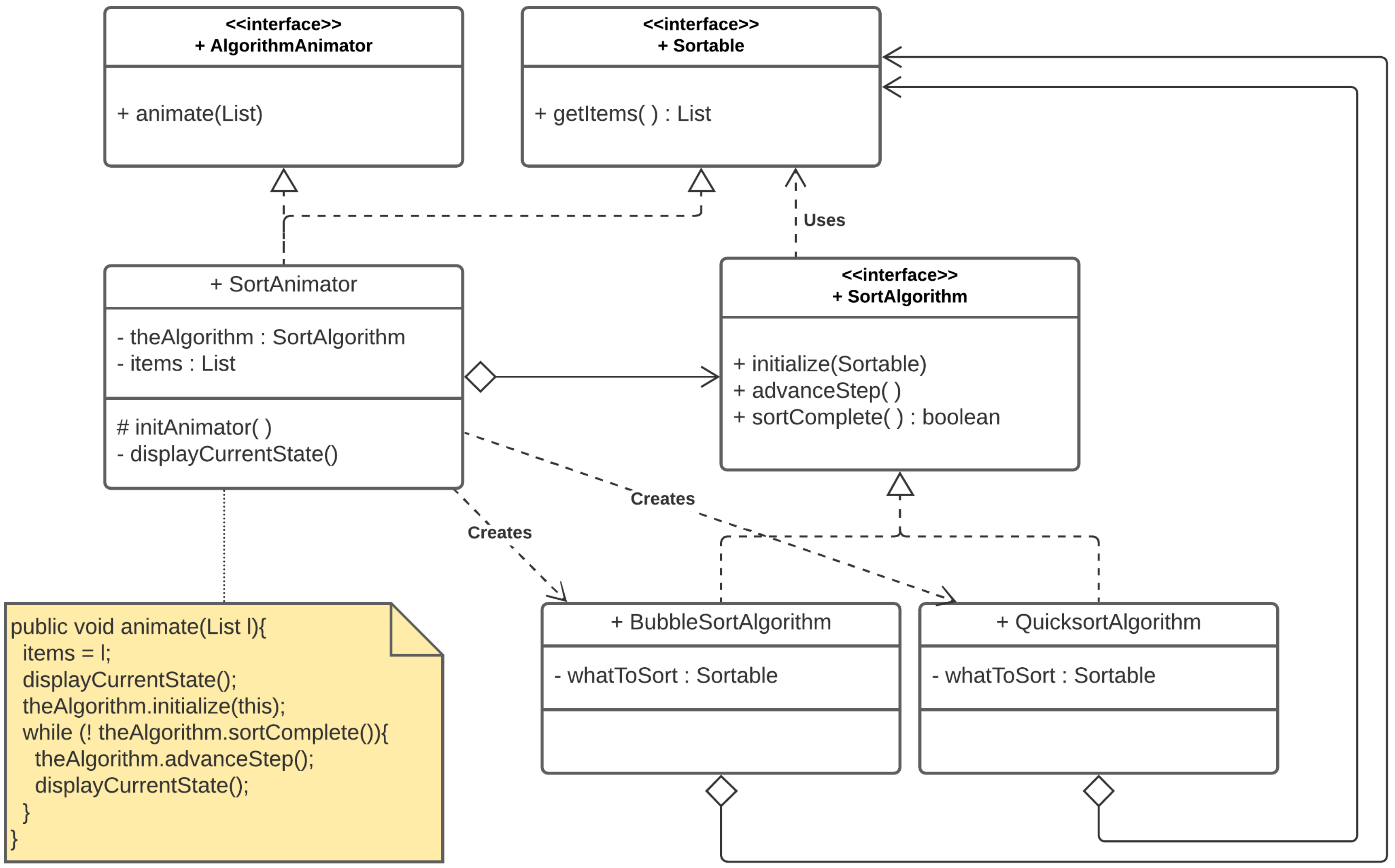
Agenda

- Design pattern: Strategy cont.

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



```
public void animate(List l){
  items = l;
  displayCurrentState();
  theAlgorithm.initialize(this);
  while (! theAlgorithm.sortComplete()){
    theAlgorithm.advanceStep();
    displayCurrentState();
  }
}
```

Creating instances of concrete algorithms

```
public class SortAnimator implements
  AlgorithmAnimator, Sortable{
  private SortAlgorithm theAlgorithm;
  private List items;
  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

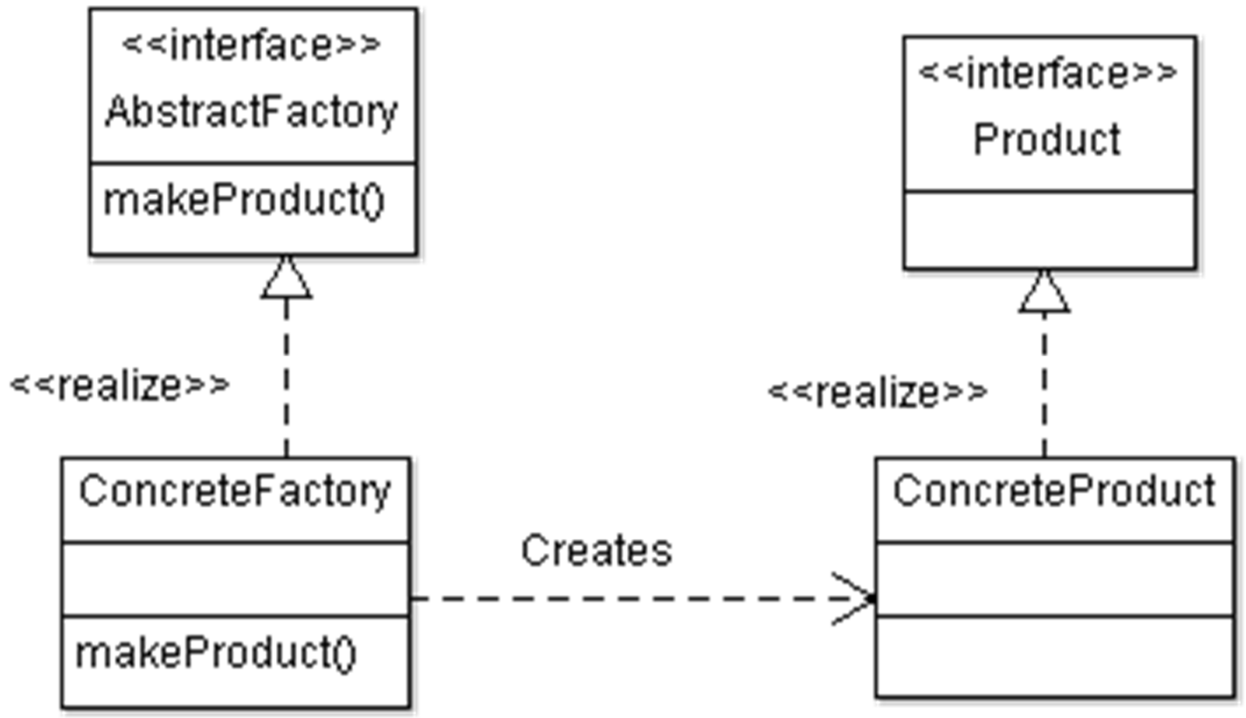
Factory pattern

- **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

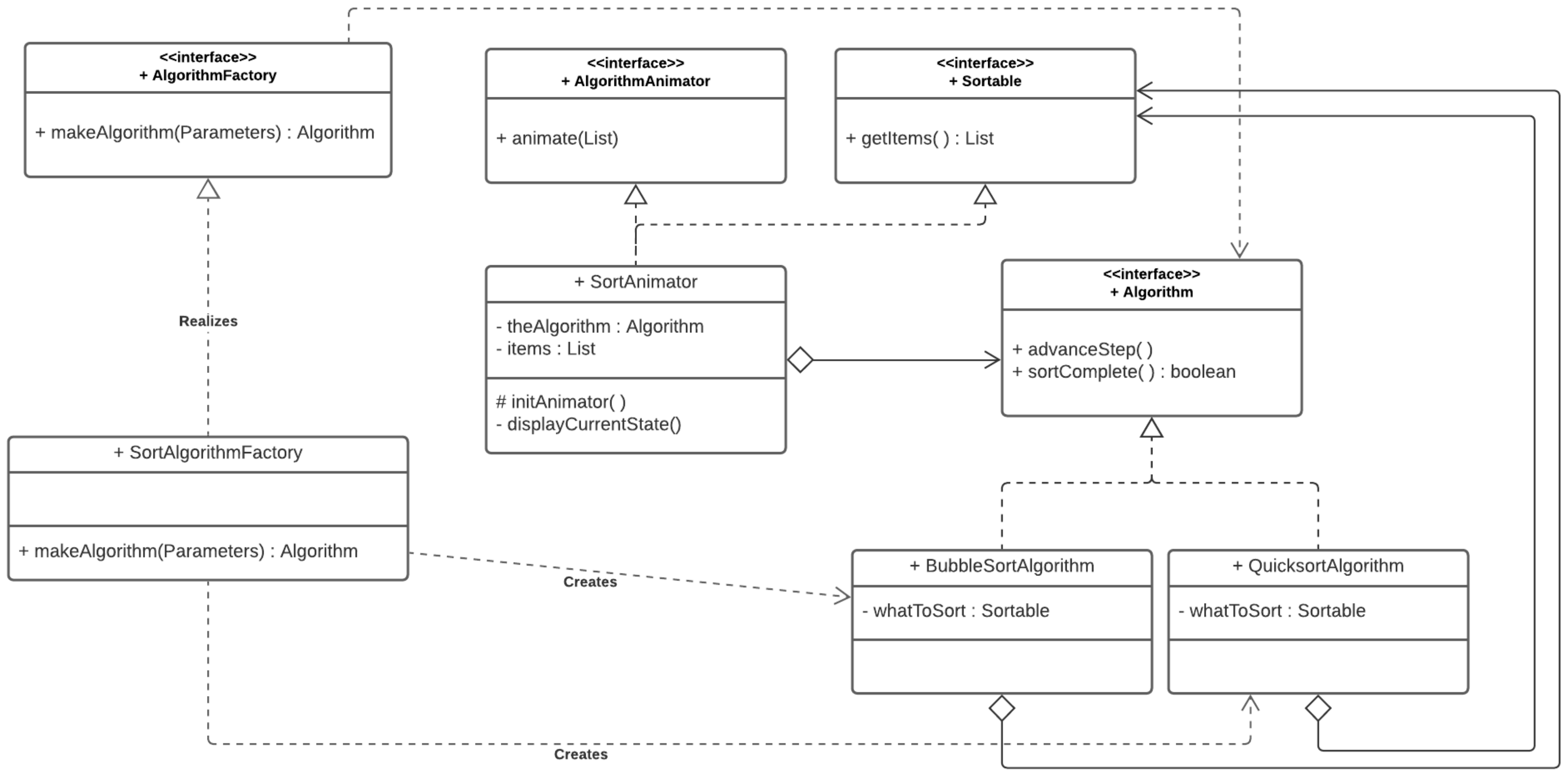
- **Product** – Defines an interface of objects the factory will create
- **ConcreteProduct** – Implements the Product interface
- **AbstractFactory** – Defines a factory method that returns an object of type Product
- **ConcreteFactory** – Overrides the factory method to return an instance of ConcreteProduct

Factory UML



<<interface>>
AbstractFactory
makeProduct()
<<interface>>
Product
<<realize>>
<<realize>>
ConcreteFactory
makeProduct()
Creates
ConcreteProduct

Example UML



<<interface>>
+ AlgorithmFactory
+ makeAlgorithm(Parameters) : Algorithm
<<interface>>
+ AlgorithmAnimator
+ animate(List)
<<interface>>
+ Sortable
+ getItems( ) : List
Realizes
+ SortAnimator
- theAlgorithm : Algorithm
- items : List
# initAnimator( )
- displayCurrentState()
<<interface>>
+ Algorithm
+ advanceStep( )
+ sortComplete( ) : boolean
+ SortAlgorithmFactory
+ makeAlgorithm(Parameters) : Algorithm
Creates
+ BubbleSortAlgorithm
- whatToSort : Sortable
+ QuicksortAlgorithm
- whatToSort : Sortable
Creates

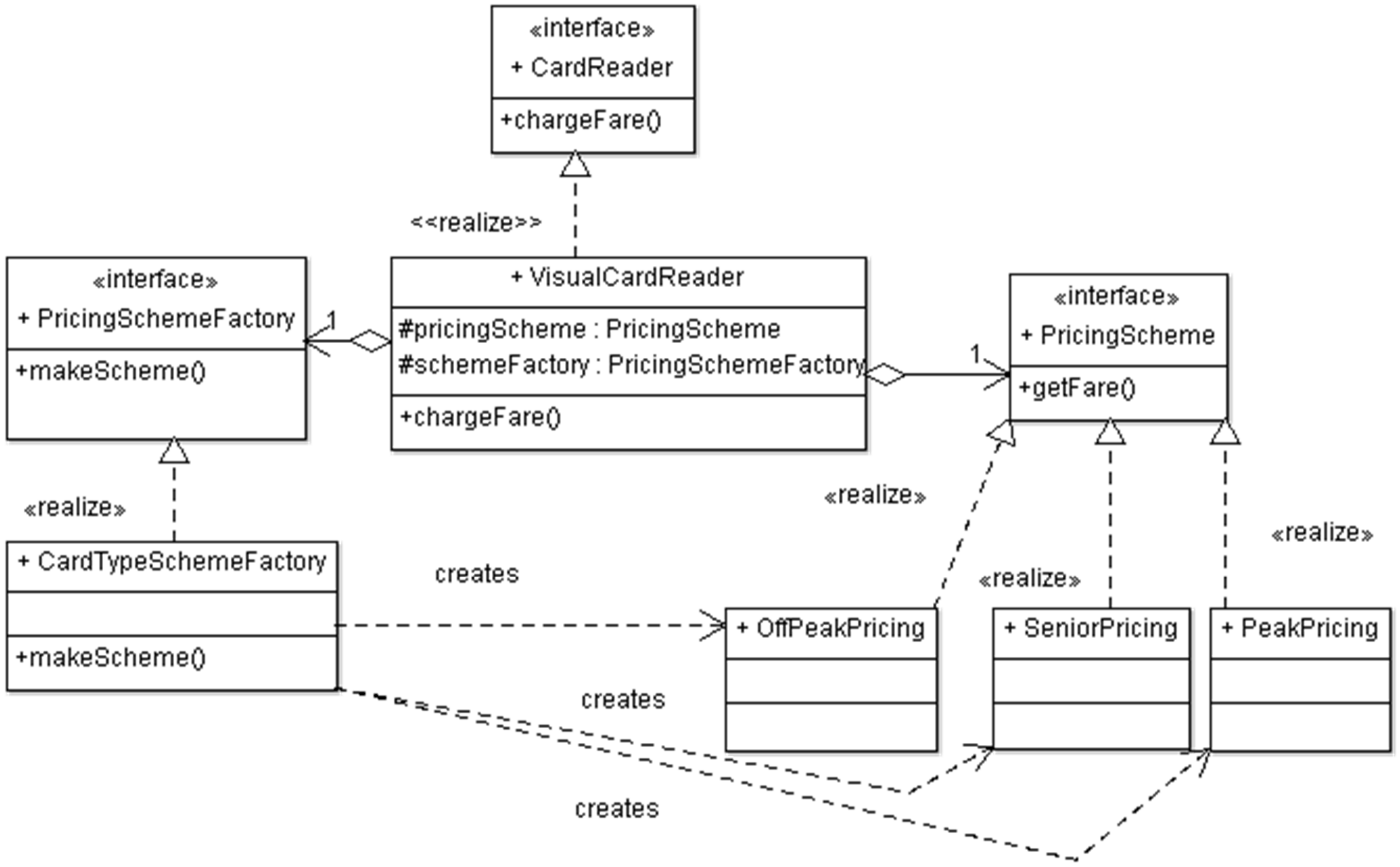
Revised SortAnimator

```
public class SortAnimator implements AlgorithmAnimator{
  private Algorithm theAlgorithm;
  private AlgorithmFactory algorithmFactory;
  protected void initAnimator(){
    String at = getParameter(“alg”);
    algorithmFactory = new SortAlgorithmFactory(this);
    theAlgorithm = algorithmFactory.makeAlgorithm(at);
  }
}
```

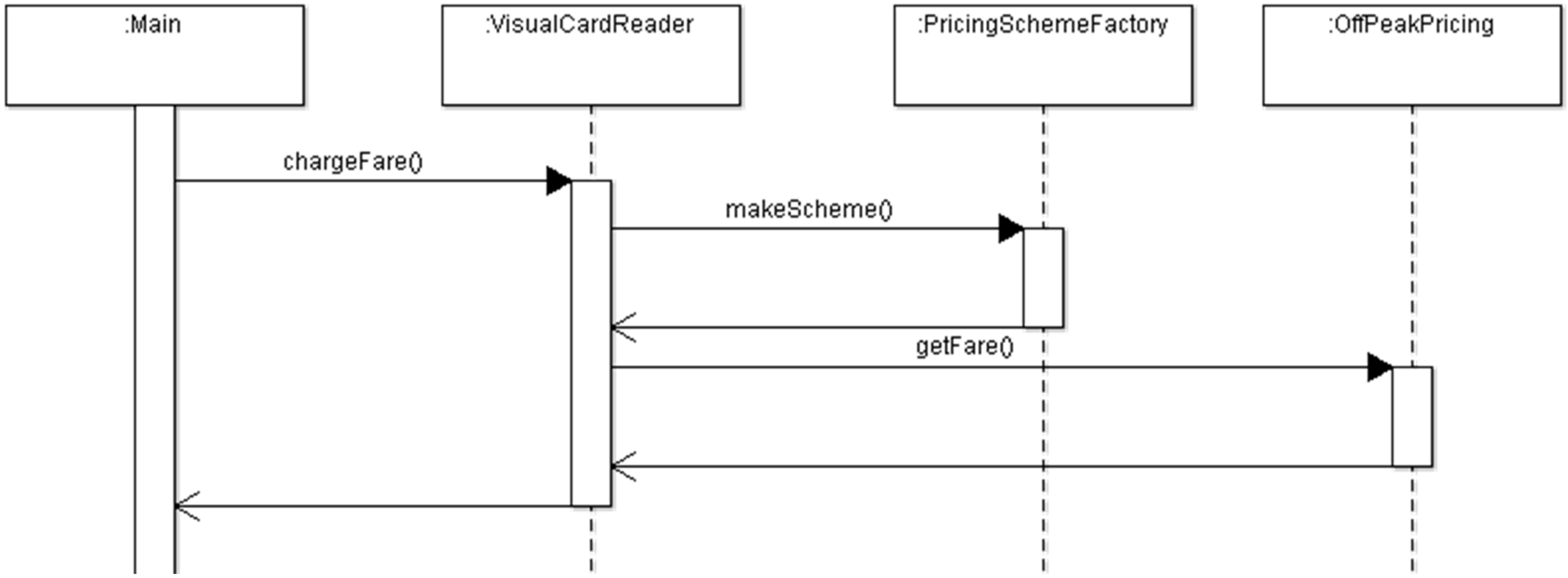
Card reader problem

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.**