Lecture 2

Dealing with Runtime Errors

Bad References

Useful when you know

- A class's specifications
- But not how to implement it completely.

```
public abstract class Shape {
 private double area;
 public abstract void computeArea();
 public double getArea() {
   return area;
 }
 // more fields and methods
```

Catching Exceptions

```
public class Except1 {
   public static void main(String[] s) {
      Student s1 = null;
      try {
         s1.toString();
      } catch (NullPointerException ne) {
         System.out.println(" Null Pointer Exception on s1.
\n Continue(yes/no)?");
         Scanner myScanner = new Scanner(System.in);
         String answer = myScanner.next();
         if (answer.equals("no")) {
            System.exit(0);
         }
         myScanner.close();
```

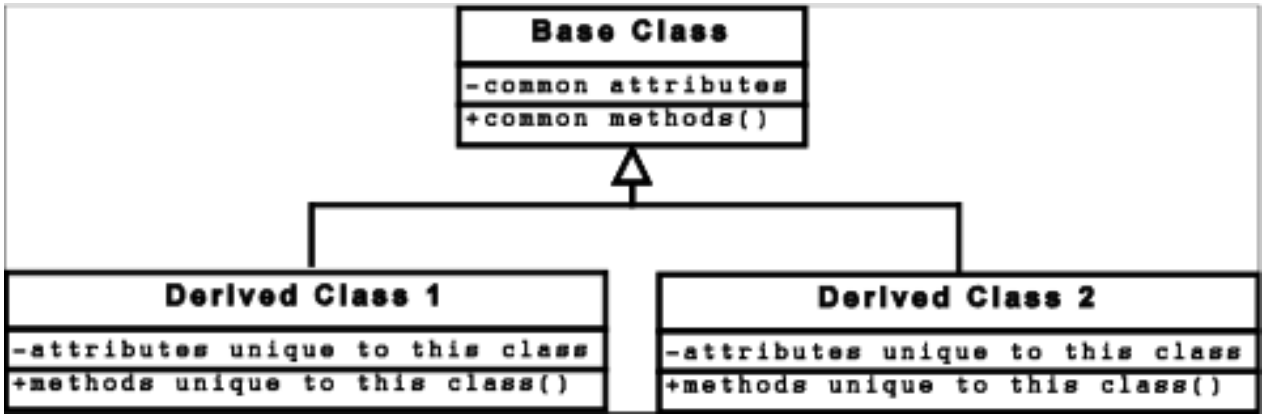
Exceptions Can Be Propagated

```
class Course {
   private String name;
   public String getName() {
       return name;
   }
}
class Section {
   private Course course;
   private String id;
   public Section(Course course) {
       this.course = course;
```

Inheritance

Base/Super and Derived/Sub Classes

- Base class/superclass
- Derived class/subclass



Inheritance Example

```
public class Product {
 private String company;
 private double price;
 private int quantitySold;
 public Product(String company, double price) {
   this.company = company;
   this.price = price;
 }
 public void sell() {
   quantitySold++;
 }
 public void setPrice(double newPrice) {
   price = newPrice;
 }
 public String toString() {
   return "Company: " + company +  " price: " +       price + " qty sold " + quantitySold;
 }
}
```

Inheritance Example

```
public class Television extends Product {
 private String model;
 public Television(String model, String manufacturer,
                                 double price) {
   super(manufacturer, price);
   this.model = model;
 }
 public String toString() {
   return super.toString() + " model: " + model;
 }
}
```

Inheriting from an Interface

```
public class Television extends Product {
 private String model;
 public Television(String model, String manufacturer,
                                 double price) {
   super(manufacturer, price);
   this.model = model;
 }
 public String toString() {
   return super.toString() + " model: " + model;
 }
}
```

Inheritance Example

```
public interface I {
   // details of I
}
public class A implements I {
   //code for A
}
public class B implements I {
   //code for B
}
I i1 = new A(); // i1 holds a reference to an object of type
A
I i2 = new B(); // i2 holds a reference to an object of type
```

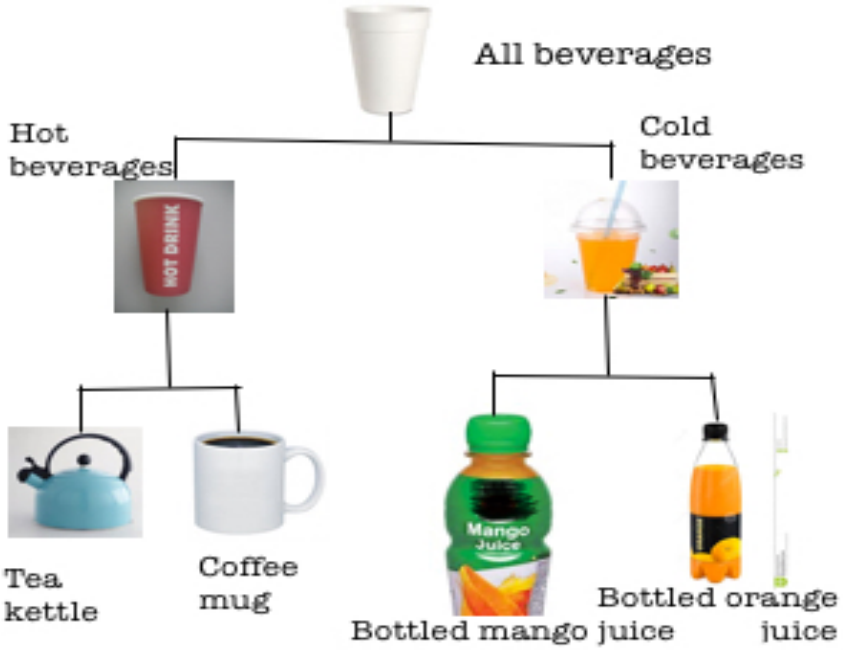
Polymorphism and Dynamic Binding

Graphics from publicly available files on the Internet

Through a Practical Example

Imagine a Beverage Shop

It offers beverages

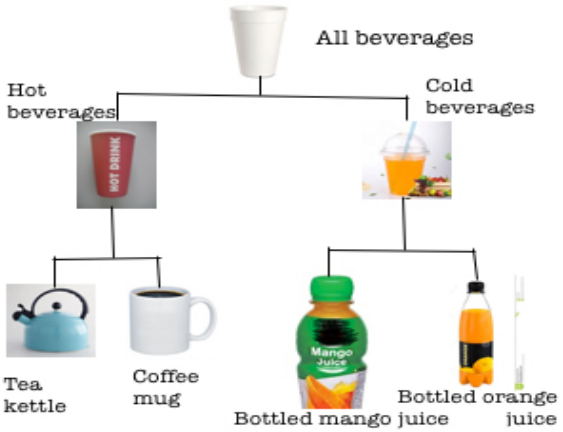


Remember this correspondence

Each product (eg a specific bottle of orange juice is like an object

Each type of product is like a class (Coffee, Beverage, etc.)

Each container is like a reference variable



We will now play a simple game

Some beverage is poured into a certain beverage container.

Without seeing the drink, but simply by looking at its container, what could you say about the type of drink in the container?

Would you be able to tell what kind of beverage it is, simply by drinking from the beverage's container, even if you are blindfolded?

Would you be able to tell whether a beverage is hot or cold by drinking from the beverage's container, even if you are blindfolded?

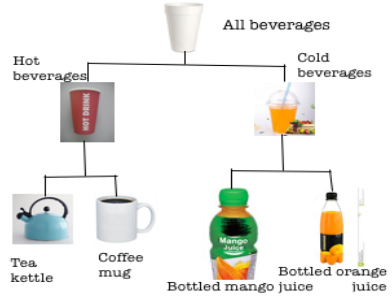
Guess what this holder/cup could contain

Coffee?

Tea?

Mango juice?

Orange juice?



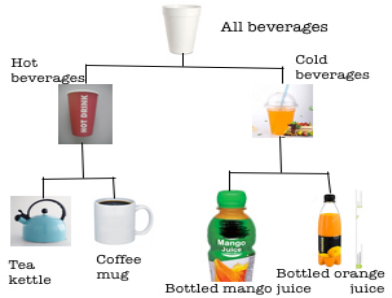
Guess what this holder/cup could contain

Coffee? ⊘

Tea? ⊘

Mango juice? ✓

Orange juice? ✓



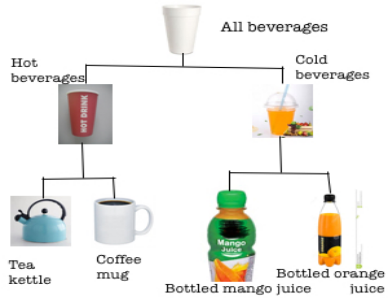
Guess what this bottle could contain

Coffee?

Tea?

Mango juice?

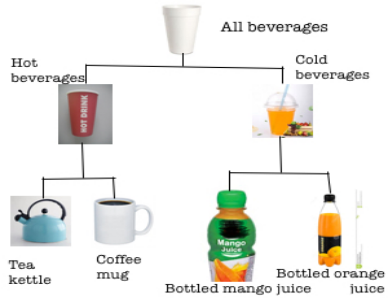
Orange juice?



Could this be

hot? ✓

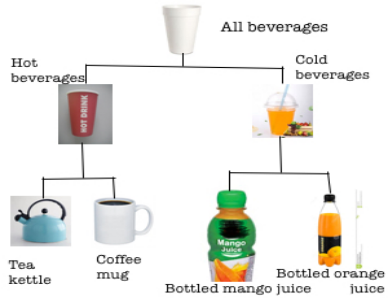
cold? ⊘



Could this be

hot? ✓

cold? ✓



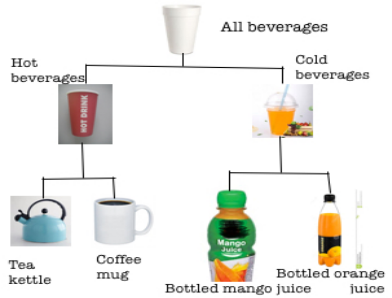
What could it taste like?

Coffee? ✓

Tea? ✓

Mango juice? ✓

Orange juice? ✓



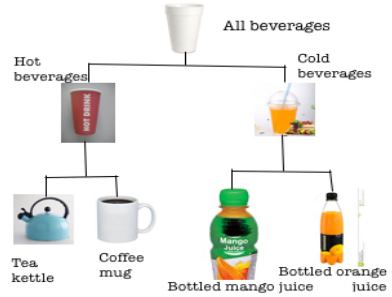
Relate these to object-oriented concepts

Some beverage is poured into a certain beverage holder. (**An object is stored into a reference variable**.)

Can you tell without seeing the drink, but simply by looking at its container, what type of drink it is? (**Related to polymorphism**.)

Should you be able to tell what kind of beverage it is, simply by drinking from a holder, even if you are blindfolded? (**Related to dynamic binding**.)

Class Declarations



```
public class Beverage {
}
```

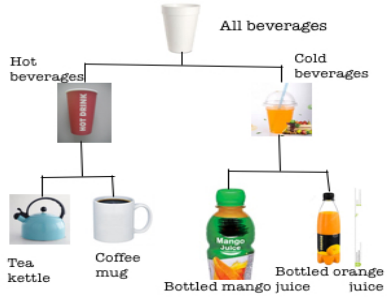

```
public class HotBeverage extends Beverage {
}
```

```
public class ColdBeverage extends Beverage
{
}
```

Class Declarations

```
public class Coffee extends HotBeverage{
}
```

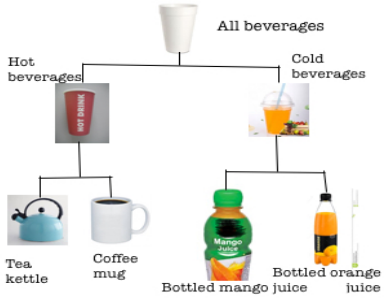
```
public class Tea extends HotBeverage{
}
```



Class Declarations


```
public class MangoJuice extends ColdBeverage{
}
```


```
public class OrangeJuice extends  ColdBeverage{
}
```



The Beverage Holders as Variables

Beverage beverageHolder;

HotBeverage hotBeverageHo

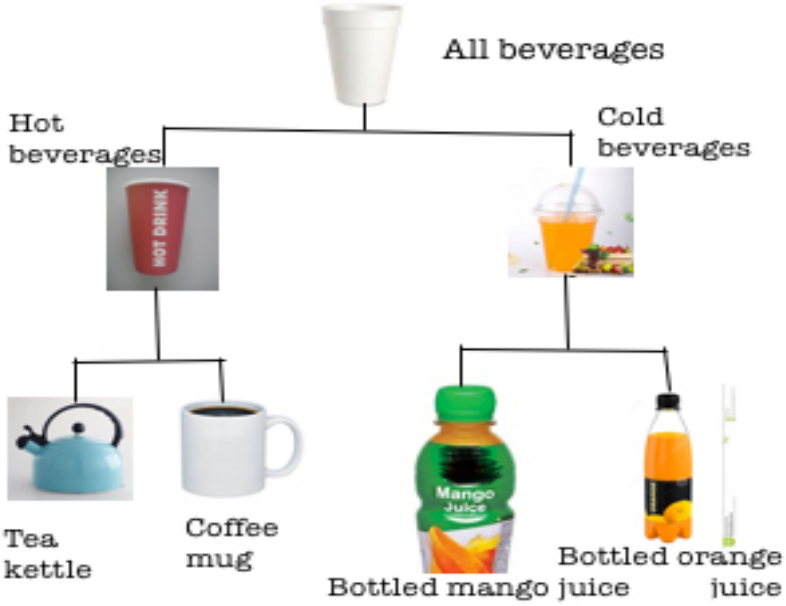
ColdBeverage coldBeverageH

Coffee coffeeMug;

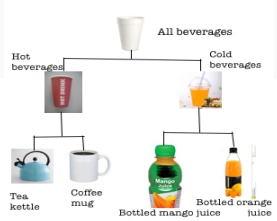
Tea teaKettle;

OrangeJuice orangeJuiceBott

MangoJuice mangoJuiceBottl



What would you expect a reference to contain?



Coffee?

Beverage beverageHolder = new Coffee(8);

Tea?

Beverage beverageHolder = new Tea(6);

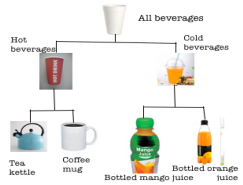
Mango juice?

Beverage beverageHolder = new MangoJuice(20);

Orange juice?

Beverage beverageHolder = new OrangeJuice(18);

Can you tell the temperature by drinking but not seeing the beverage?



coldBeverageHolder.temperature( )

hot?

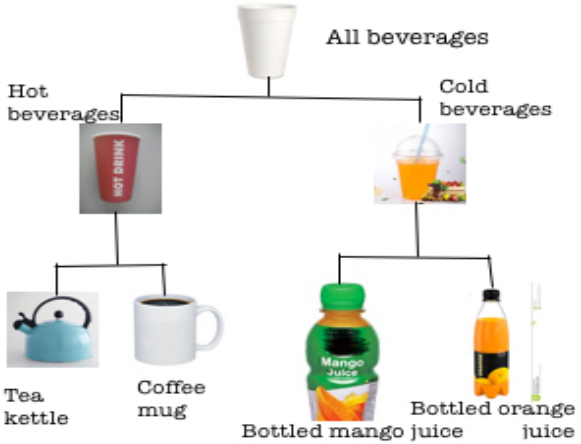
cold?

dynamic binding supports real-life scenarios

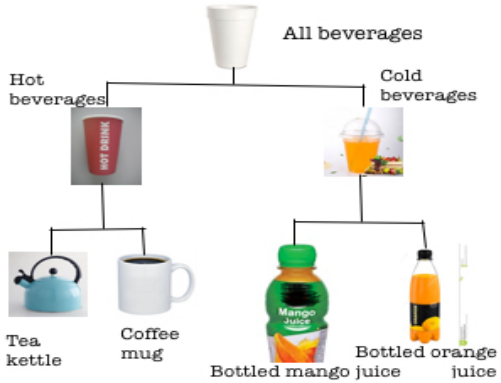
You don't have to know the object type to execute the overriding method just as you don't have to see the beverage when you drink it to see if it is cold or hot.

New Drink!

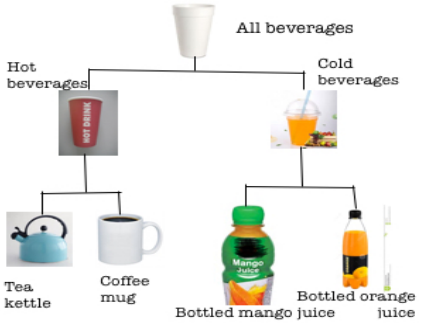
```
public class Cola extends ColdBeverage{
}
```



Mango Juice could be organic



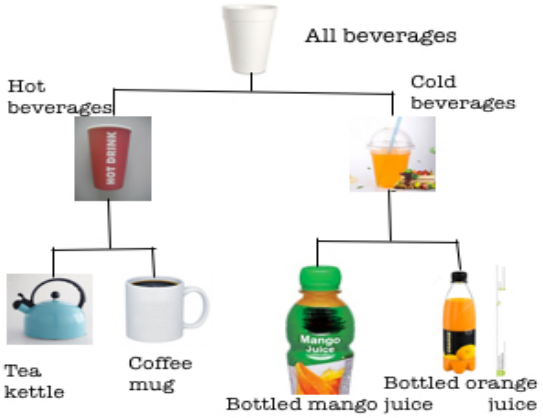
Is Mango Juice Organic?




```
MangoJuice mangoJuiceBottle = new MangoJuice(20);
System.out.println(mangoJuiceBottle.isOrganic());
```

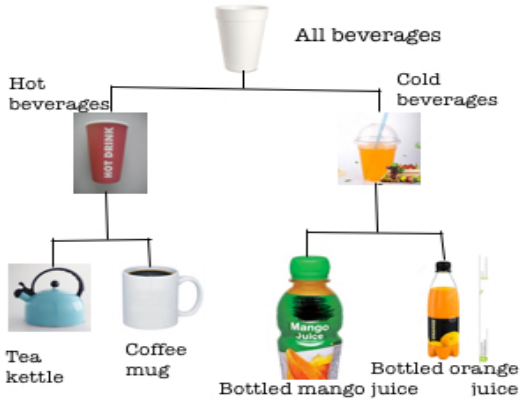
Can you do the following on either of these?

Is the drink organic?



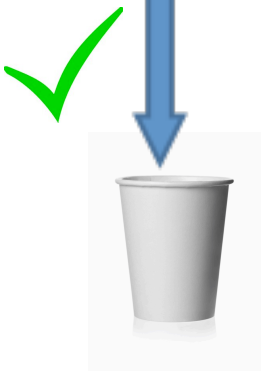
So in Java…

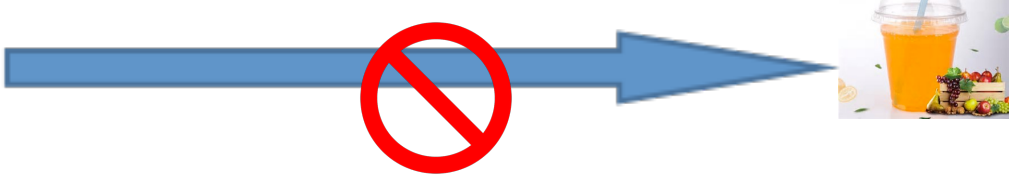
```
ColeBeverage coldBeverageHolder = new
MangoJuice(20);

System.out.println(coldBeverageHolder
.isOrganic());
```

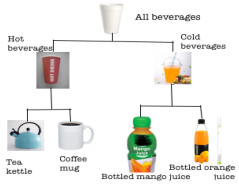


Pouring the contents of a holder to another

beverageHolder = coldBeverageHolder;



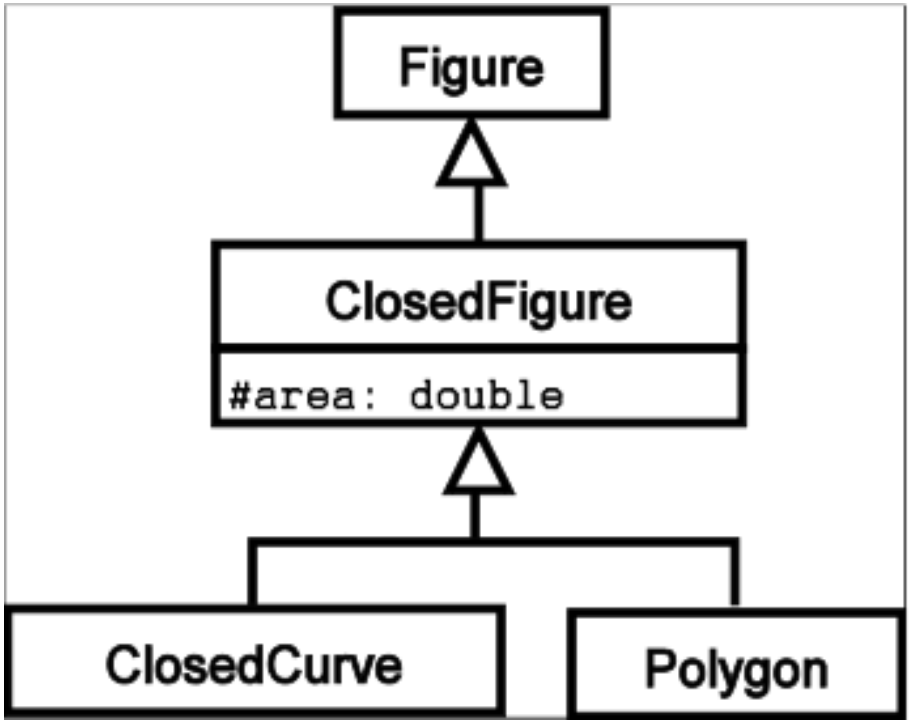




coldBeverageHolder = beverageHolder;

Protected Members

Protected Fields and Methods



Protected Fields and Methods

Protected Fields and Methods

Run-Time Type Identification

Run Time Type Identification (RTTI)

Need to know the number of Circle objects in a ShapeList collection.

int circleCount(Shapelist shapeList)

The method iterates through all the items in the collection, check which ones are of type Circle}. If so, increment a counter

Need a mechanism to detect if a given Shape object is a Circle.

One approach: Have a method in the Shape class that returns true when a Shape object is a Circle

Drawbacks:

- defeats the purpose of having dynamic binding
- inelegant if we had a large hierarchy.

RTTI Using Java Reflection

```
Shape shape;
// code to create a Shape object
// and store its reference in shape
if (shape.getClass().getName().equals("Circle")) {
 // take appropriate action
}
```

RTTI Using Java Reflection

The compiler cannot check for typographical errors in the string against which we are checking the name.

```
Shape shape;
// code to create s Shape object
// and store its reference in shape
if (shape.getClass().getName().equals("circle")) {
 // take appropriate action
}
```

RTTI Using the instanceof Operator

```
Shape shape;
// code to create s Shape object
// and store its reference in shape
if (shape instanceof Circle) {
 // take appropriate action
}
```

is always a better alternative to using getClass().getName()

RTTI Using Downcasting

```
double computeTax(Investment investment) {
 double amount;
 try {
   Deposit deposit = (Deposit) investment;
   amount = deposit.getInterest();
   // code for computing tax on amount
 } catch(ClassCastException cce) {
   try {
     Stock stock = (Stock) investment;
     amount = stock.getInterest();
     // code for computing tax on amount
   } catch(ClassCastException cce) {
     cce.printStackTrace();
   }
 }
 // return tax
}
```

The Object Class

Can Hold References to Any Type

Use of equals

```
String id = "id1";
Book book1 = new Book("title1", "author1", id);
System.out.println(book1.equals(id)); // call 1
System.out.println(id.equals(book1)); // call 2
```

equals is an Equivalence Relation

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is *transitive*: for any non-null reference values x, y, and z,
  if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

There is more…

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The equals Method

```
public boolean equals(Object anObject) {
   if (anObject == null) {
      return false;
   }
   if (!(anObject instanceof Student)) {
      return false;
   }
   Student student = (Student) anObject;
   return student.name.equals(name) &&
 student.address.equals(address);
}
```

Other Important Object Methods

- toString()
- clone()

Garbage Collection

Other Important Object Methods

An object is considered to be garbage if it cannot be reached, directly or indirectly through any reference.

```
Student s1 = new Student(``John", ``X1");
StudentList slist = new StudentList();
slist.add(s1);
```

Iterator

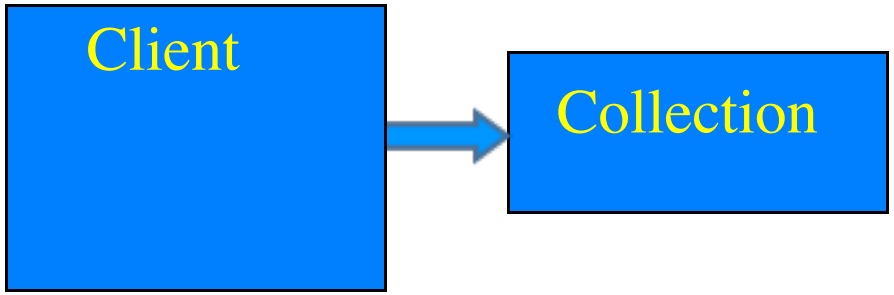
Changes in a Collection

ListImplementation1:

size() returns the number of

Varying the Collection

Clients will break if changes to the collection's implementation modify the collection's interface as well.



A Solution: Iterator

- The supported interface does not change.
- Collection traversal implemented by employing a special type of object, independent of the internal organization of the collection.
- Every collection is required to return an iterator object, which provides standard methods to traverse the collection. (Iterable in the JDK)

myCollection.iterator()

returns an iterator object.

Components

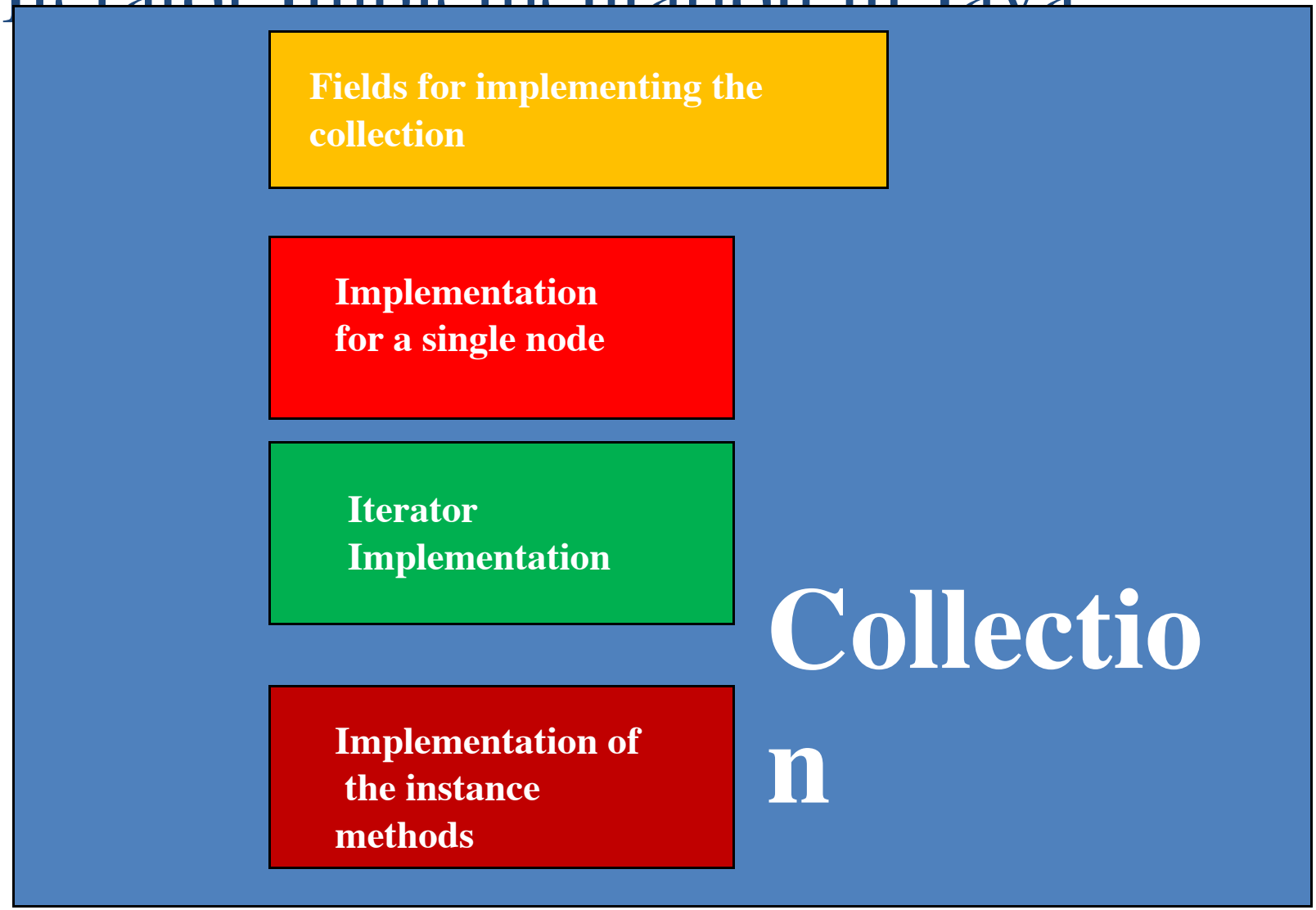
- Collection: an interface that allows the usual operations to add and delete objects, plus the method iterator() that returns an iterator object.
- Iterator: an interface that supports the operations hasNext() and next() described above.
- Implementation of the Collection interface: They could use arrays, linked lists, hash tables, etc.
- Implementation of the Iterator: must cooperate with the code in the Collection's implementation
- Client code that uses the collection.

Using the Iterator in Java

```
Collection collection = new LinkedList();
collection.add("Element 1");
collection.add(new Integer(2));
for (Iterator iterator = collection.iterator();
   iterator.hasNext(); ) {
   System.out.println(iterator.next());
}
```

Iterator Implementation

Iterator Implementation in Java



Iterator Implementation in Java

```
public class LinkedQueue implements Queue,
   Iterable {
   // fields for managing the queue
   private class Node {
   // code to implement a single node
   }
   private class QueueIterator implements Iterator {
   // code for QueueIterator
   }
   public Iterator iterator() {
     return new QueueIterator();
   }
   // Queue methods
```

Fields in LinkedQueue

```
private Node head;
private Node tail;
private int numberOfElements;
```

A Node in LinkedQueue

```
private class Node {
   private Object data;
   private Node next;
   private Node(Object object, Node next) {
     this.data = object;
     this.next = next;
   }
   public Object getData() {
     return data;
   }
   public void setNext(Node next) {
     this.next = next;
   }
   public Node getNext() {
     return next;
   }
 }
```

Deleting an Element from LinkedQueue

```
public Object remove() {
  if (head == null) {
    return null;
  }
  Object value = head.getData();
  head = head.getNext();
  if (head == null) {
    tail = null;
  }
  numberOfElements--;
  return value;
}
```

Iterator Implementation

```
private class QueueIterator implements Iterator {
   private  Node current = head;
   public boolean hasNext() {
        return current != null;
   }
   public Object next() {
         if (hasNext()) {
             Object data = current.getData();
             current = current.getNext();
             return data;
         }
         throw new NoSuchElementException();
  }
}
```

Using an Iterator

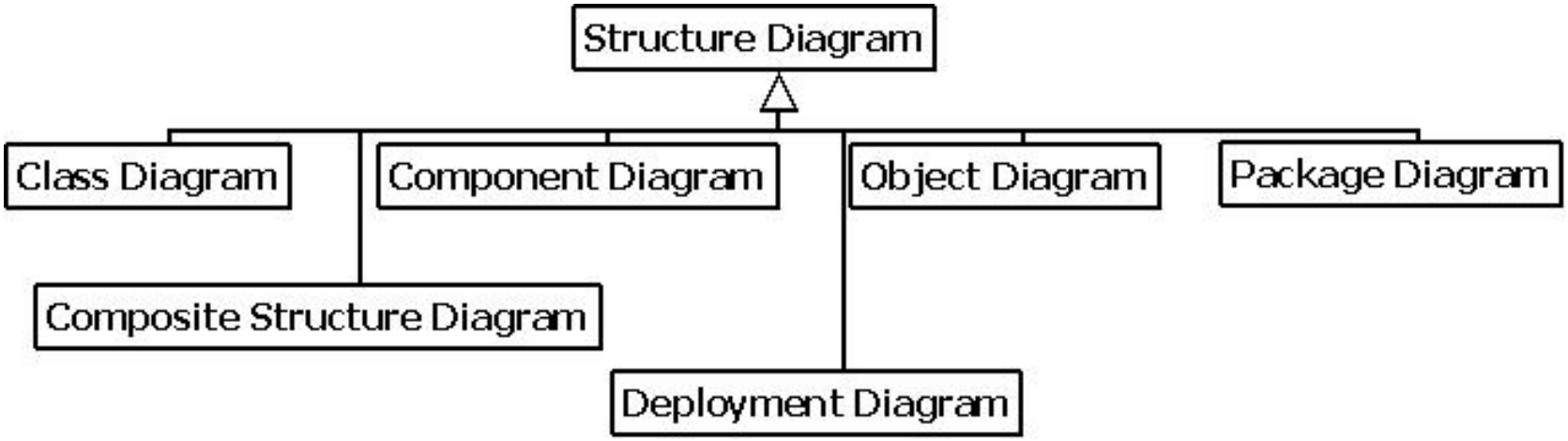
```
Collection collection = new LinkedList();
collection.add("Element 1");
collection.add(new Integer(2));
for (Iterator iterator1 = collection.iterator(); iterator1.hasNext(); ) {
  Integer int1 = iterator1.next();
  count = 0;
  for (Iterator iterator2 = collection.iterator(); iterator2.hasNext(); ) {
      Integer int2 = iterator2.next();
      if (int1.equals(int2)) {
        count++;
      }
  }
  System.out.println(int1 + count);
}
```

Unified Modeling Language

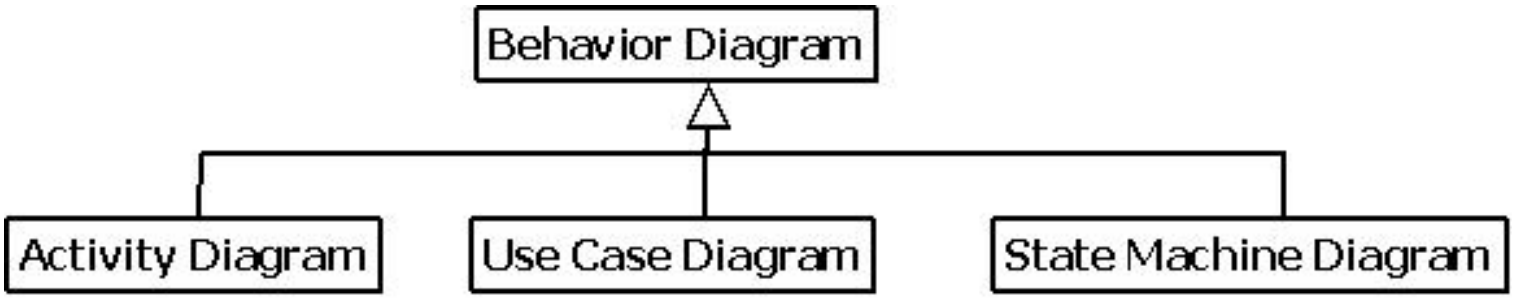
Unified Modeling Language (UML)

- Structure diagrams: show the static architecture of the system irrespective of time.
- Behavior diagrams: depict the behavior of a system or business process.
- Interaction diagrams: show the methods, interactions, and activities of the objects.

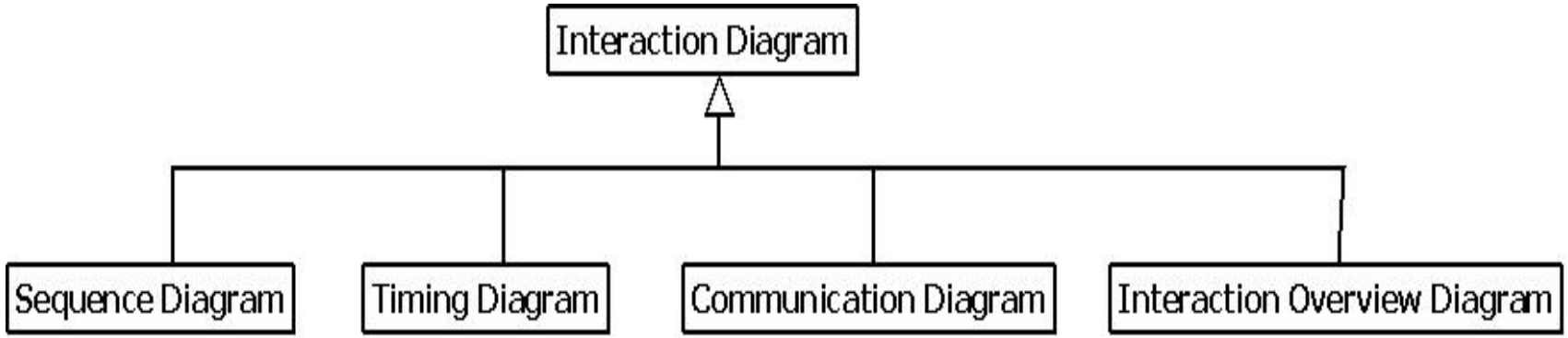
Types of UML Diagrams



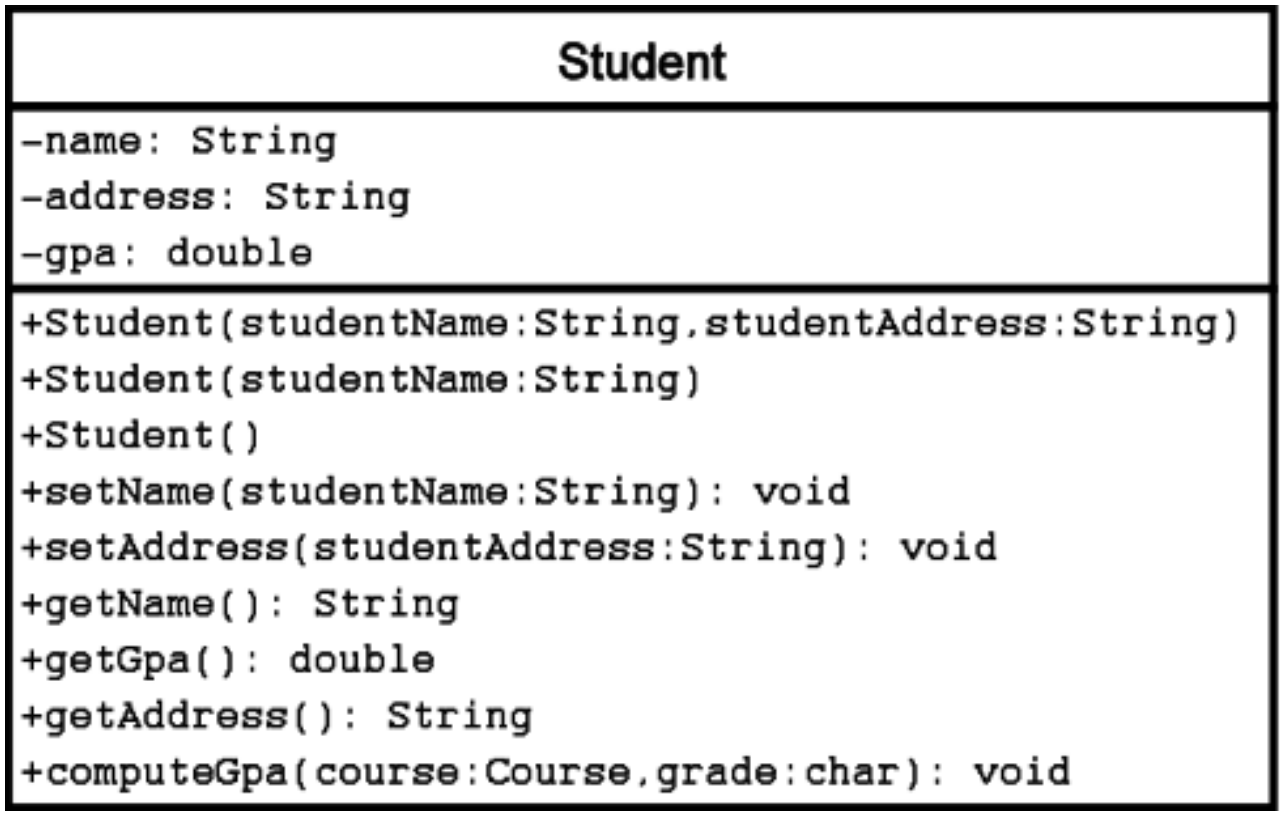
Types of UML Behavior Diagrams



Types of UML Interaction Diagrams



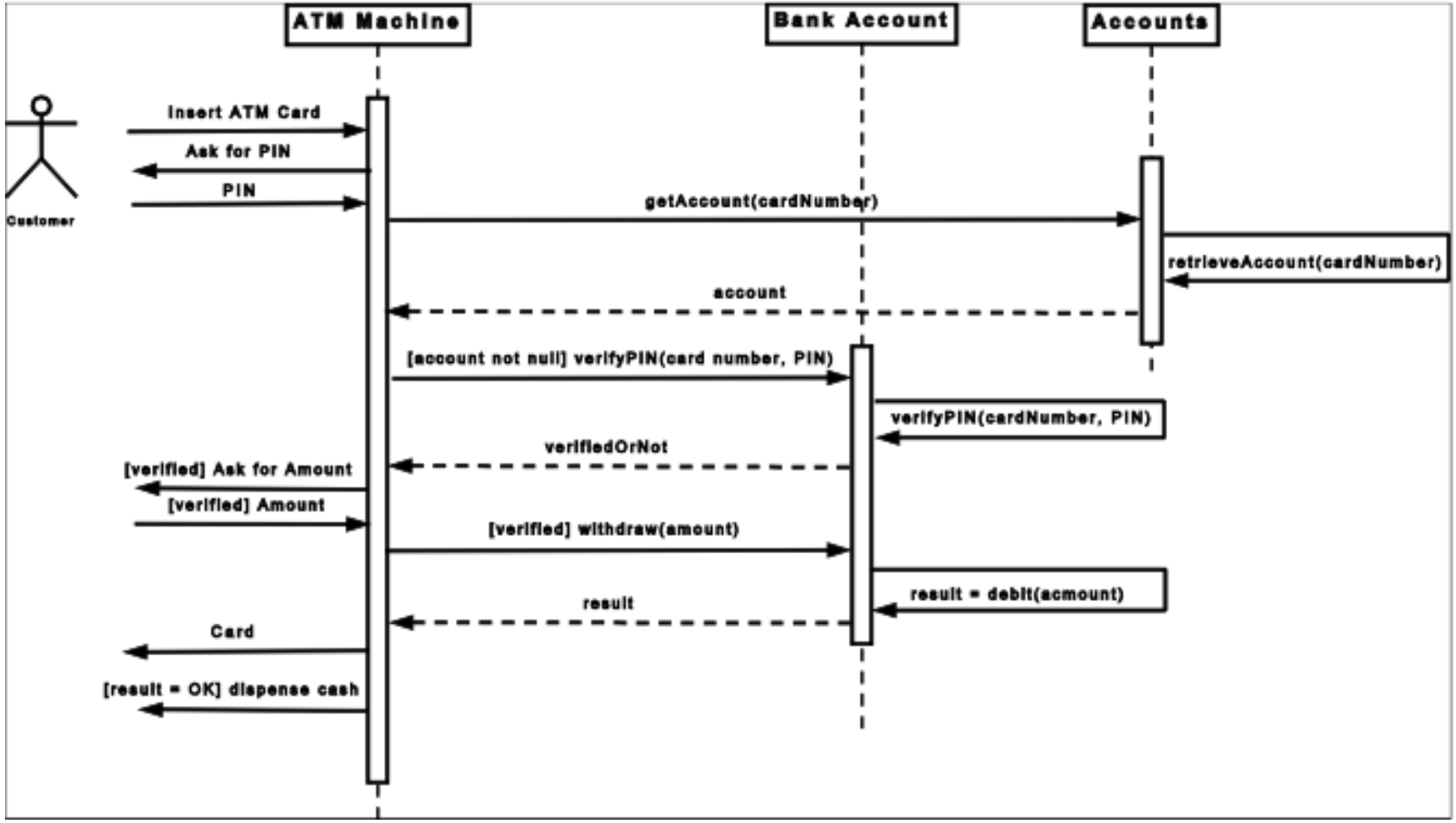
Class Diagram Example



Use-Case Example

Actions performed by the actor	Responses from the system
1) Inserts Debit Card into the “Insert Card” slot.	
	2) Asks for the PIN Number.
3) Enters the PIN Number.	
	4) Verifies the PIN. If the PIN is invalid, displays an error and goes to Step 8. Otherwise, asks for the amount.
5) Enters the amount.	
	6) Verifies that the amount can be withdrawn. If not, displays an error and goes to Step 8. Otherwise, dispenses the amount and updates the balance.
7) Takes the cash.	
	8) Ejects the card.
9) Takes the card.	

Sequence Diagram Example



Interfaces and their Implementation