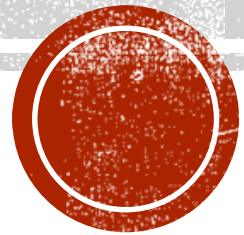
STRATEGY PATTERN



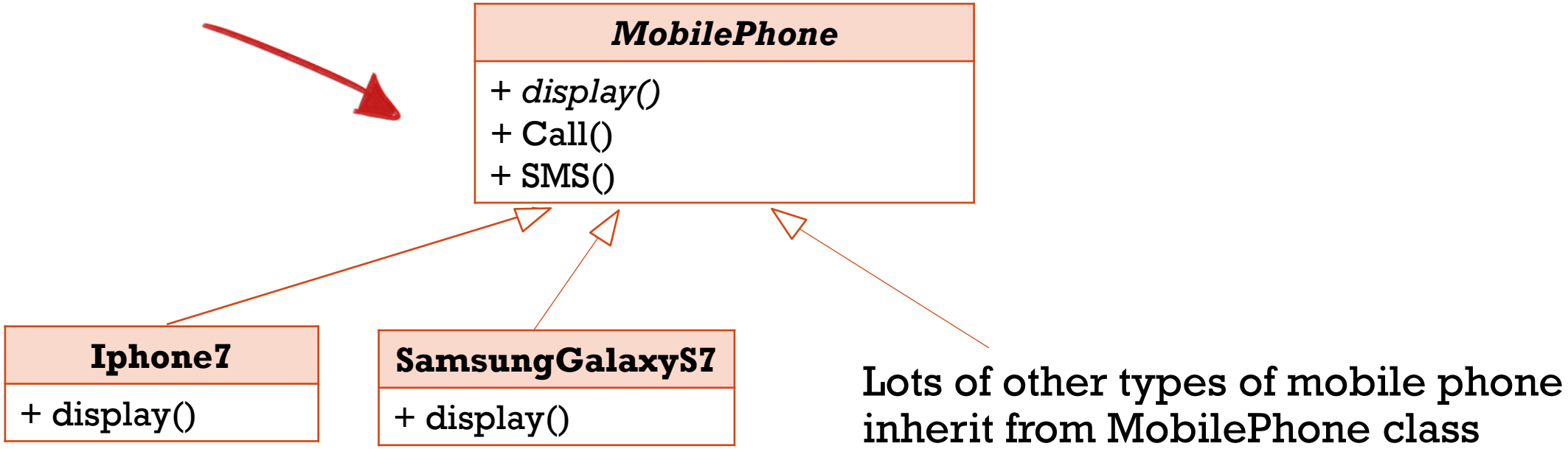
Group 4

WELCOME TO A MOBILE WORLD

- A group of designers was ordered to create a basic simulator for the mobile handset. A user can select a handset, and should be able to function whatever is supported in the handset.

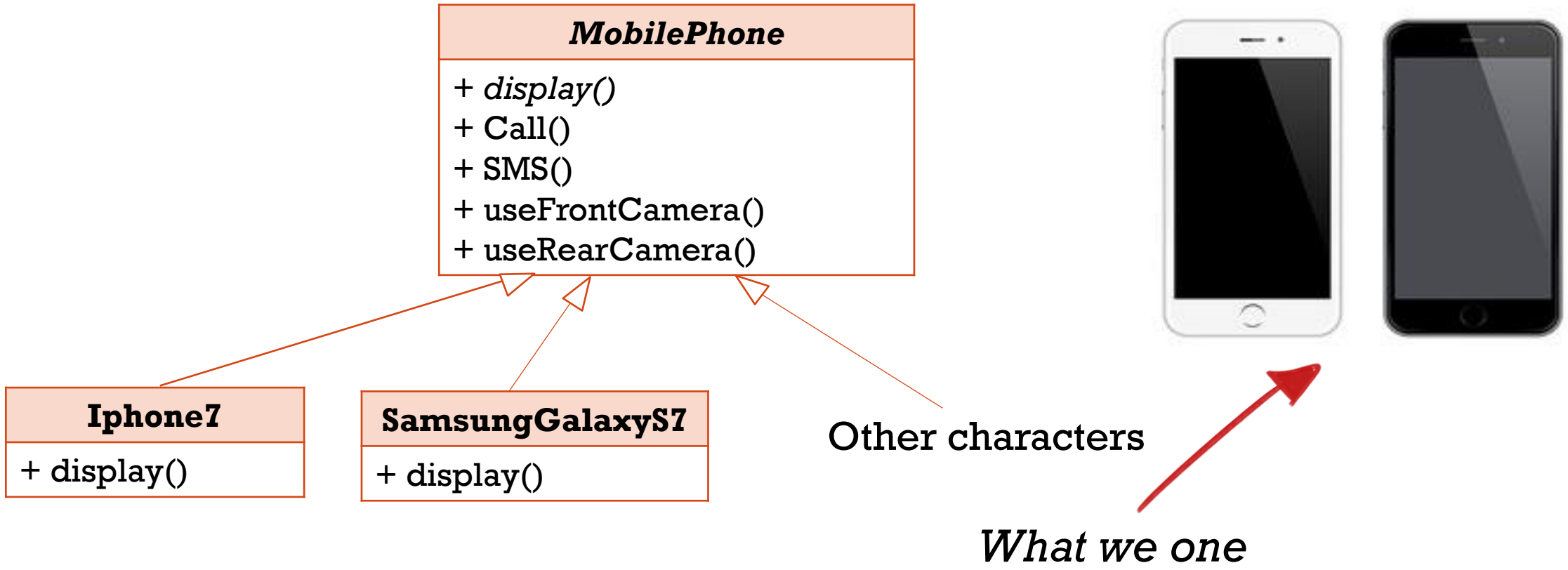
- The initial designers of the system used standard OO and created one MobilePhone superclass from which all other character types inherit.

All mobile devices have functionalities
to call and send message



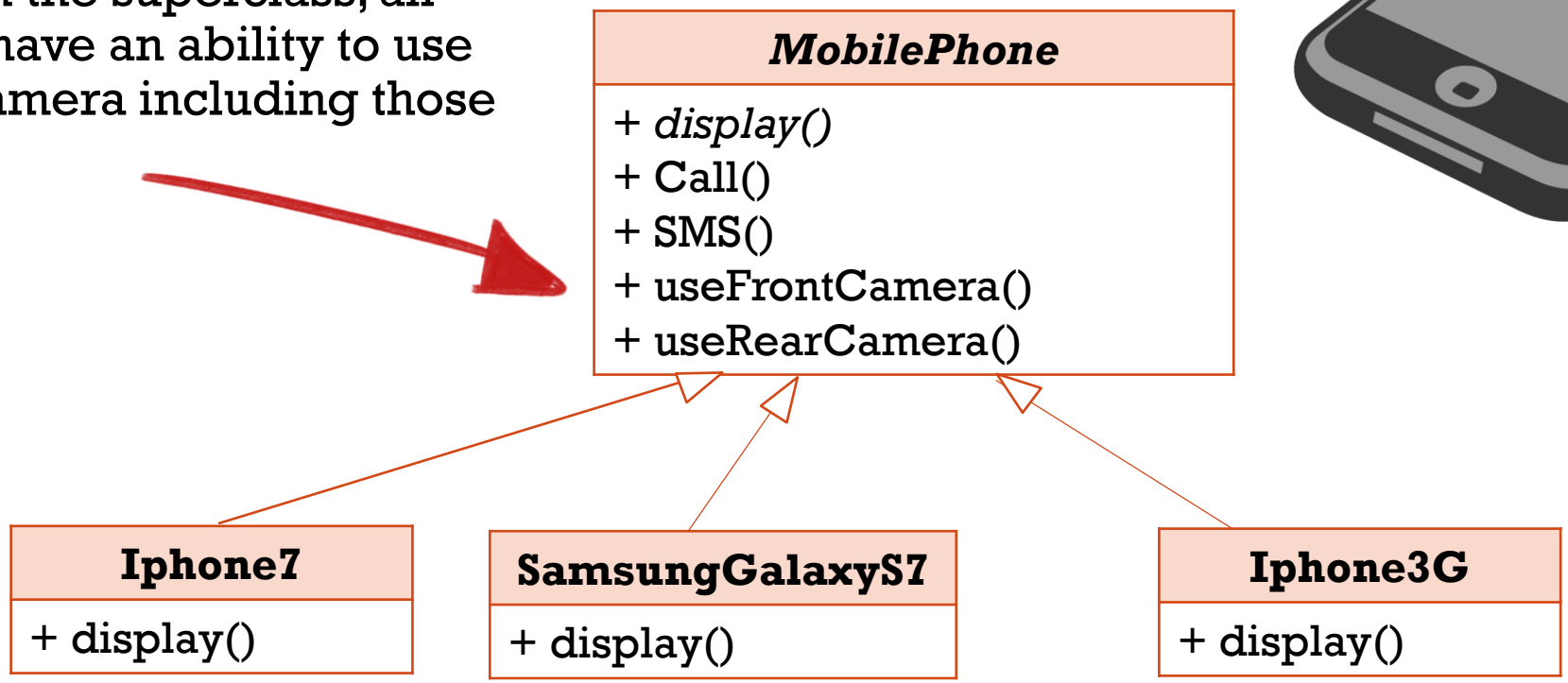
Lots of other types of mobile phone
inherit from MobilePhone class

- But now we also need the mobile phone to **take a snap**. However, the end user not only want a rear camera but also a front camera to take selfie. The designers team add two methods in the MobilePhone class and then all the mobile phones will inherit it.



- **But something went horribly wrong.**
- What if an end user uses Iphone3G and it only have a rear camera, so it is not appropriate for Iphone3G to have a front camera.

By putting useFrontCamera() and use RearCamera() in the superclass, all mobile phones have an ability to use front and rear camera including those that shouldn't



- **The designers team think about inheritance…**

Iphone3G
+ display() + useRearCamera(){//take a snap} **+ useFrontCamera(){** **// override to do nothing** **}**

Override the useRearCamera() and useFrontCamera() method

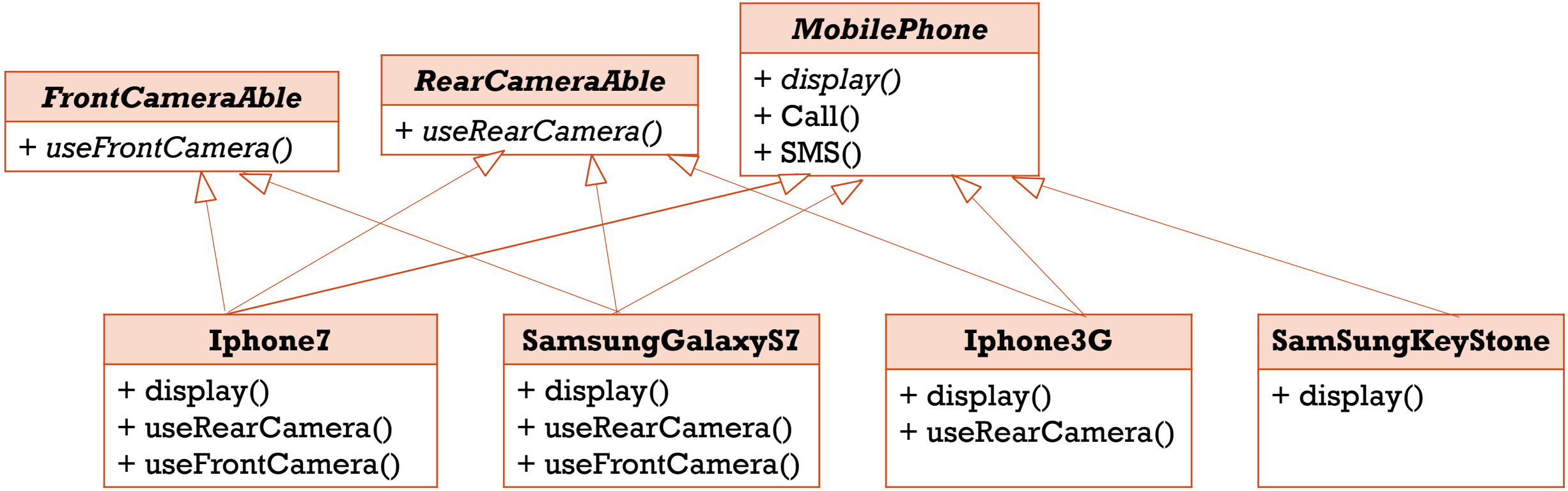
But what happens when they add this dude to the program **?**

SamsungKeystone
+ display() **+ useRearCamera(){** **//override to do nothing** **}** **+ useFrontCamera(){** **// override to do nothing** **}**

ALTERNATE SOLUTION

- The executives want to update the product every six months. Designer team realize that inheritance solution is not going to work. They know that the attributes will keep changing and they will be forced to looked at and possibly override useRearCamera() and useFrontCamera() for every new mobile devices that are added to the program… ***forever***.
- It is a maintenance nightmare for themselves, so they start redesigning their system.
- *Let' give it a try…*

How about an interface?

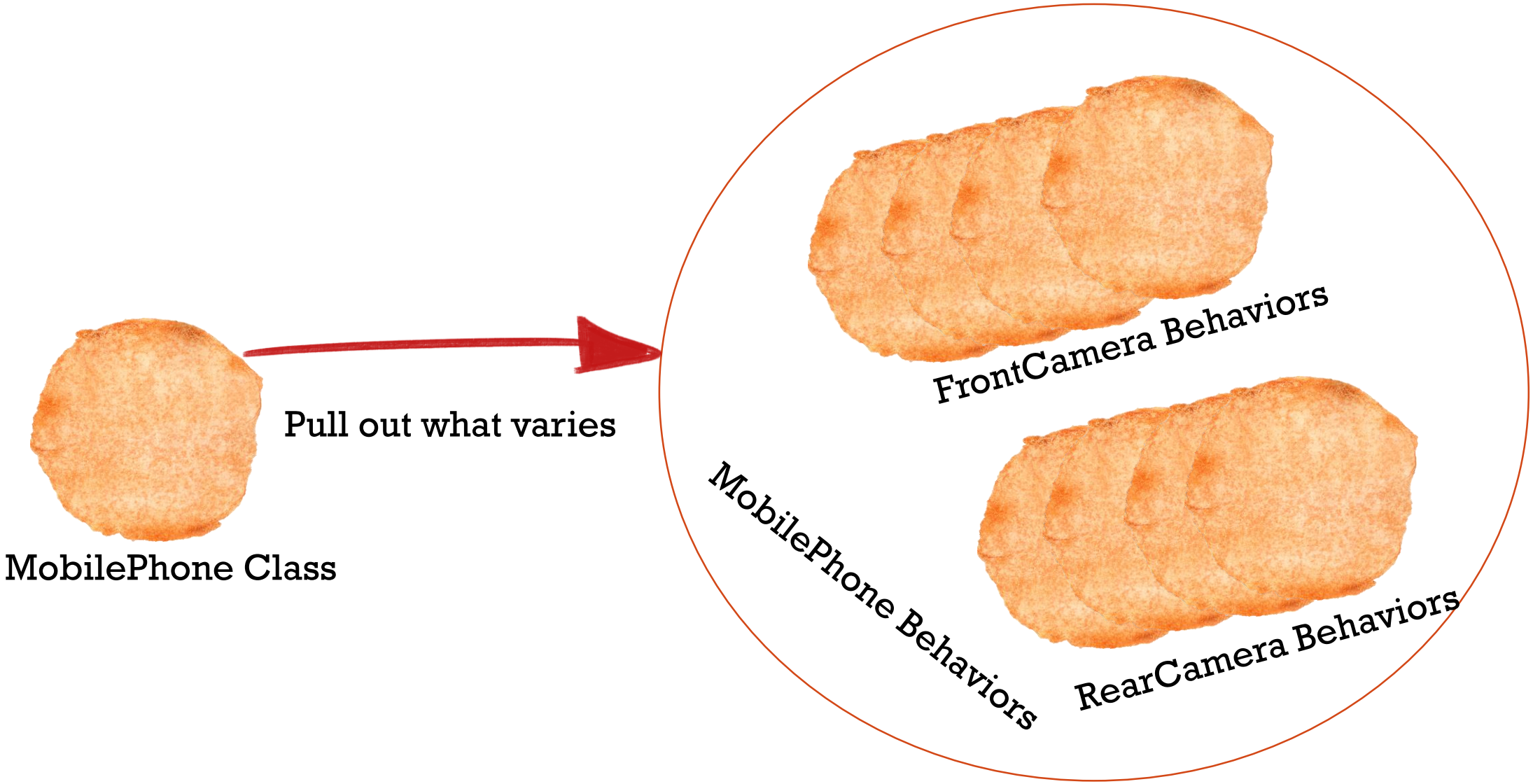


- *But there are also some potential problems with this approach…*

1. Can you say “duplicate code”, you ‘re gonna crazy when you need to make a little change to the use front camera behavior… in all tons of smart phones.
2. While having the subclasses implement FrontCameraAble and RearCameraAble solve part of the problem, it completely destroy code reuse for those behaviors.
3. What if the customers or users decide they want something else, or they want new functionality?
4. What if the company decided it is going with another database?

THE STRATEGY PATTERN

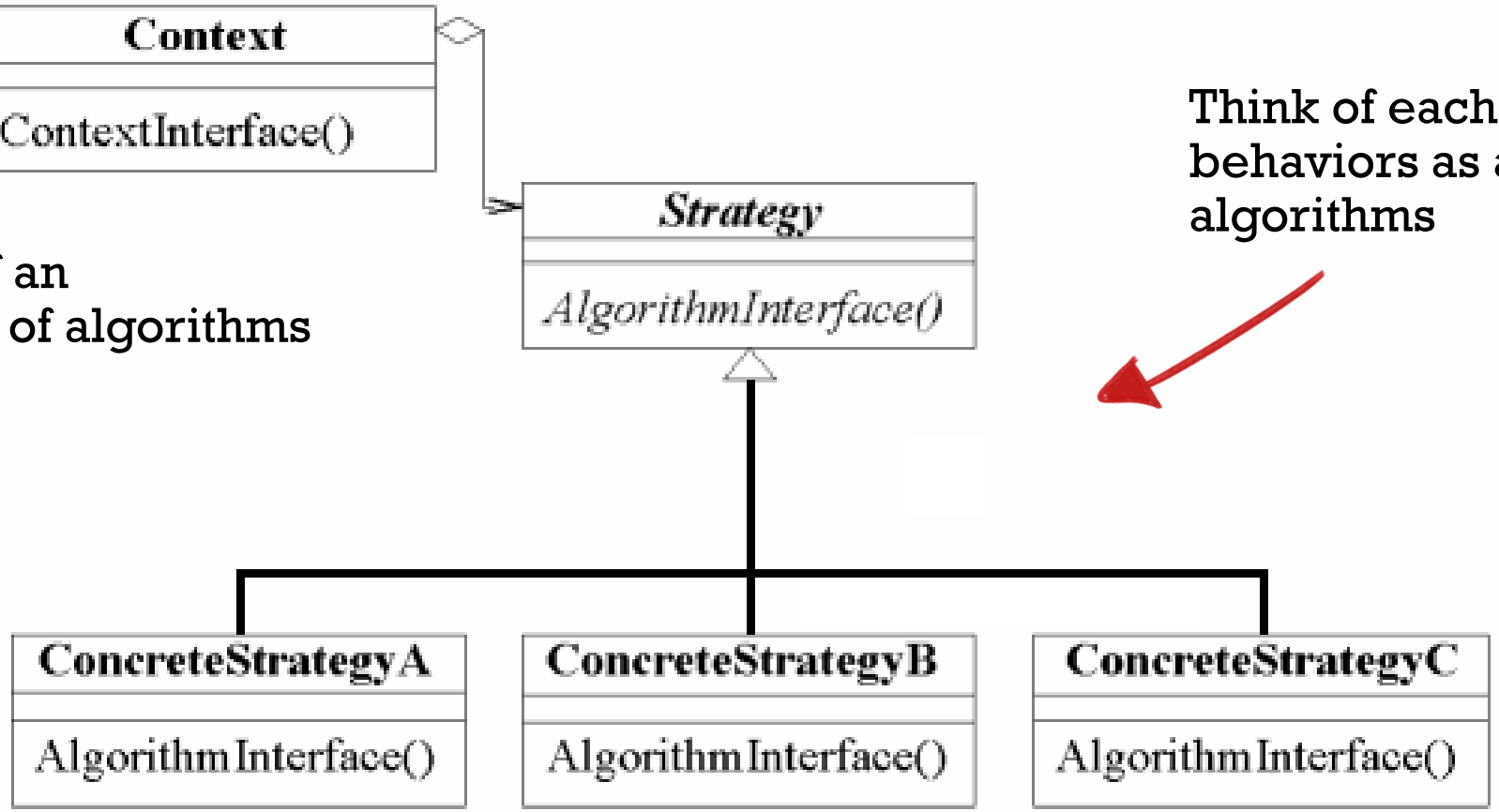
- We ‘ve seen that using inheritance hasn’t worked out very well. The FrontCameraAble and RearCameraAble interface sounded promising at first, but whenever we need to modify a behavior, we are forced to track down and change it in all different subclasses where that behavior is defined.
- Let’s implement the strategy pattern.
- We ‘ll pull both methods out of the MobilePhone class and create a new set of classes to represent each behavior.


MobilePhone Class
Pull out what varies
FrontCamera Behaviors
MobilePhone Behaviors
RearCamera Behaviors

STRATEGY PATTERN DEFINITION

- The strategy pattern defines a family of algorithms, encapsulates each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- Encapsulated separately from the validating objects, it may be used by other validating objects in different areas of the system without code duplication.

- *Let's take a look at the class diagram…*

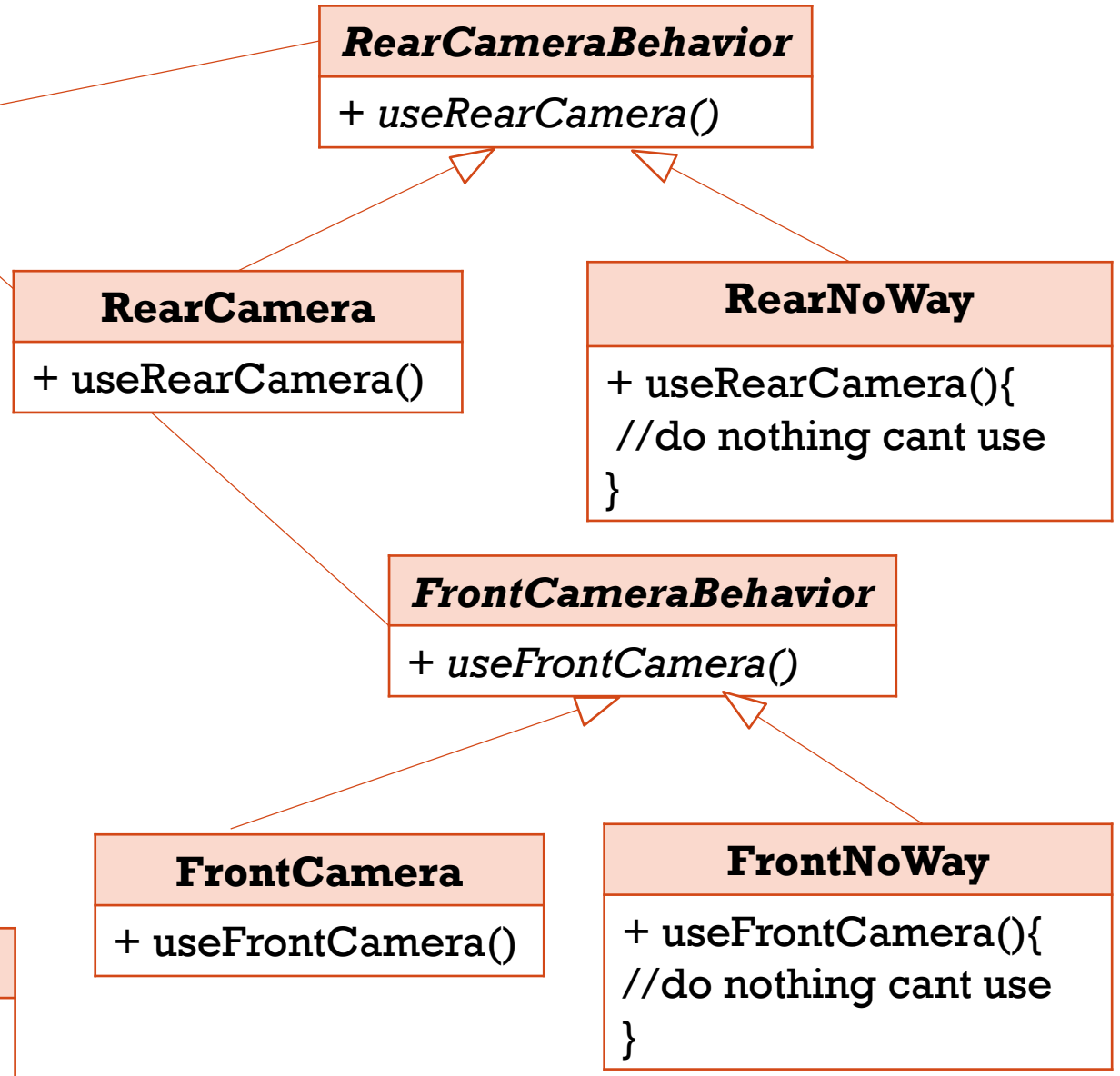


Context make use of an encapsulated family of algorithms for strategy

Think of each set behaviors as a family of algorithms

These algorithms are interchangeable

- **With this design, other type of objects can reuse our RearCamera and FrontCamera Behaviors because these behaviors are no longer hidden away in our MobilePhone classes.**
- **And we can add new behaviors without modifying any of our existing behaviors classes or touching any of the MobilePhone class that use these behaviors.**



WRITING THE MOBILE PHONE CODE

1. Coding abstract class

- Let's start with the mobile phone class, which is an abstract class.

```
class MobilePhone
{
public:
	virtual void display() = 0;
	void takeRearCamera(){
		frontCameraBehavior->useFrontCamera();
	}
	void takeFrontCamera(){
		rearCameraBehavior->useRearCamera();
	}
	MobilePhone(RearCameraBehavior* rcb, FrontCameraBehavior* fcb):
	rearCameraBehavior(rcb),frontCameraBehvior(fcb){}
private:
	RearCameraBehavior * rearCameraBehavior;
	FrontCameraBehavior * frontCameraBehavior;
};
```

- Let's implement the abstract class for Rear Camera Behavior and Front Camera Behavior.

```
class FrontCameraBehavior
{
public:
     virtual void useFrontCamera() = 0;
};
class RearCameraBehavior
{
public:
     virtual void useRearCamera() = 0;
};
```

2. Coding Mobile Phone

Now we ‘ve got our base class, let’s implement some mobile phones.

```
class Iphone7 :public MobilePhone
{
public:
        void display(){
                cout << "I am an Iphone7." << endl;
        }
        Iphone7():MobilePhone(new RearCamera(),new FrontCamera()){}
};
class SamsungKeystone :public MobilePhone
{
public:
        void display(){
                cout << "I am a samsung keystone." << endl;
        }
        SamsungKeystone():MobilePhone(new RearNoWay(),new FrontNoWay()){}
};
```

3. Coding Behaviors

It’s time to implement the concrete behaviors.

```
class RearCamera :public RearCameraBehavior
{
public:
	void useRearCamera(){
		cout << "I'm taking a photo with rear camera." << endl;
	}
	RearCamera();
};
class RearNoWay :public RearCameraBehavior
{
public:
	void useRearCamera(){
		cout << "I don't have a rear camera." << endl;
	}
	RearNoWay();
};
```

```
class FrontCamera :public FrontCameraBehavior
{
public:
	void useFrontCamera(){
		cout << "I'm taking a selfie." << endl;
	}
	FrontCamera();
};
class FrontNoWay :public FrontCameraBehavior
{
public:
	void useFrontCamera(){
		cout << "I don't have a front camera." << endl;
	}
	FrontNoWay();
};
```

4. Testing mobile phone

```
void main()
{
    MobilePhone* iphone7 = new Iphone7();
    iphone7->takeFrontCamera();
    iphone7->takeRearCamera();
}
```

- *Let's have a test…*

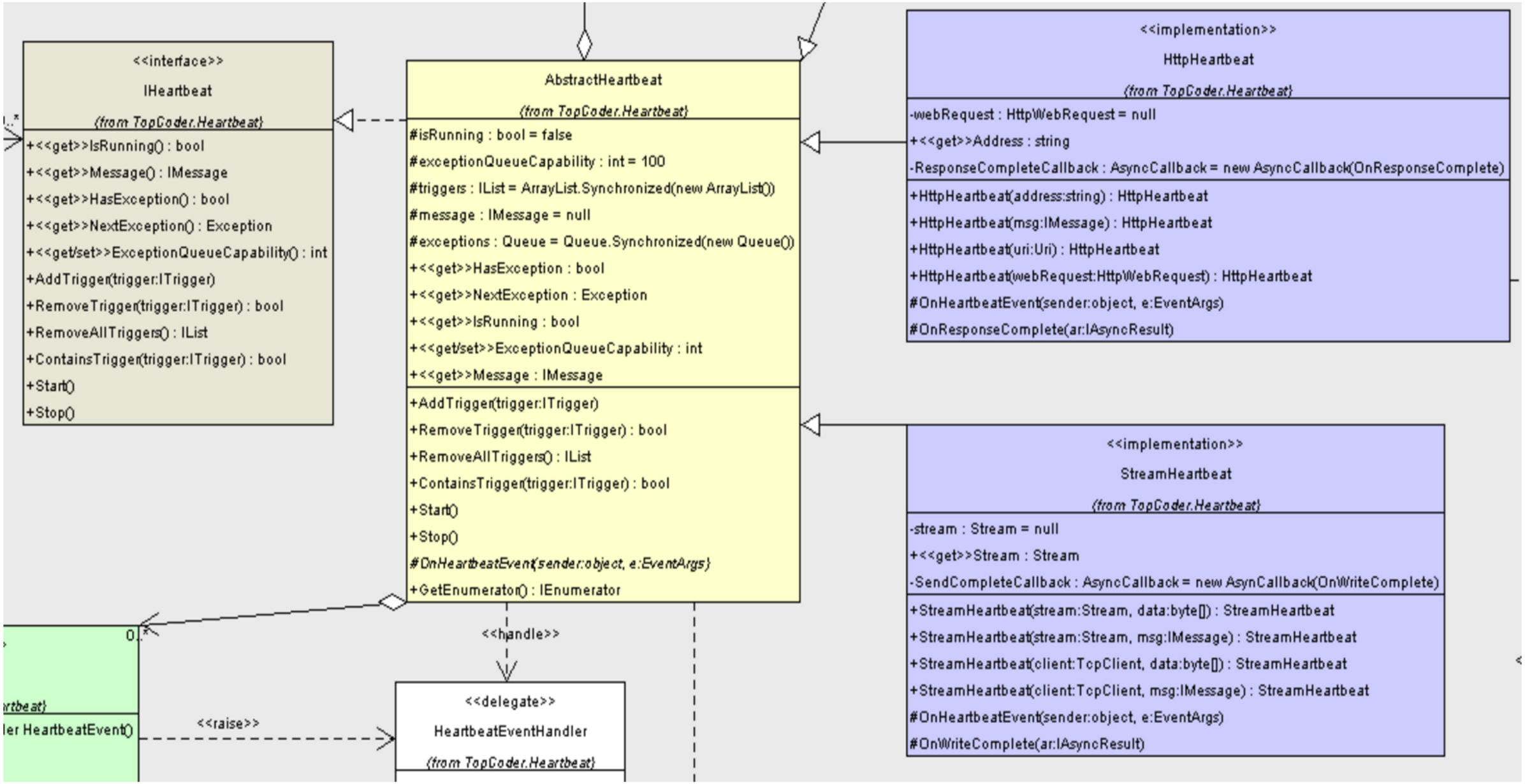
```
I'm taking a selfie.
I'm taking a photo with rear camera.
```

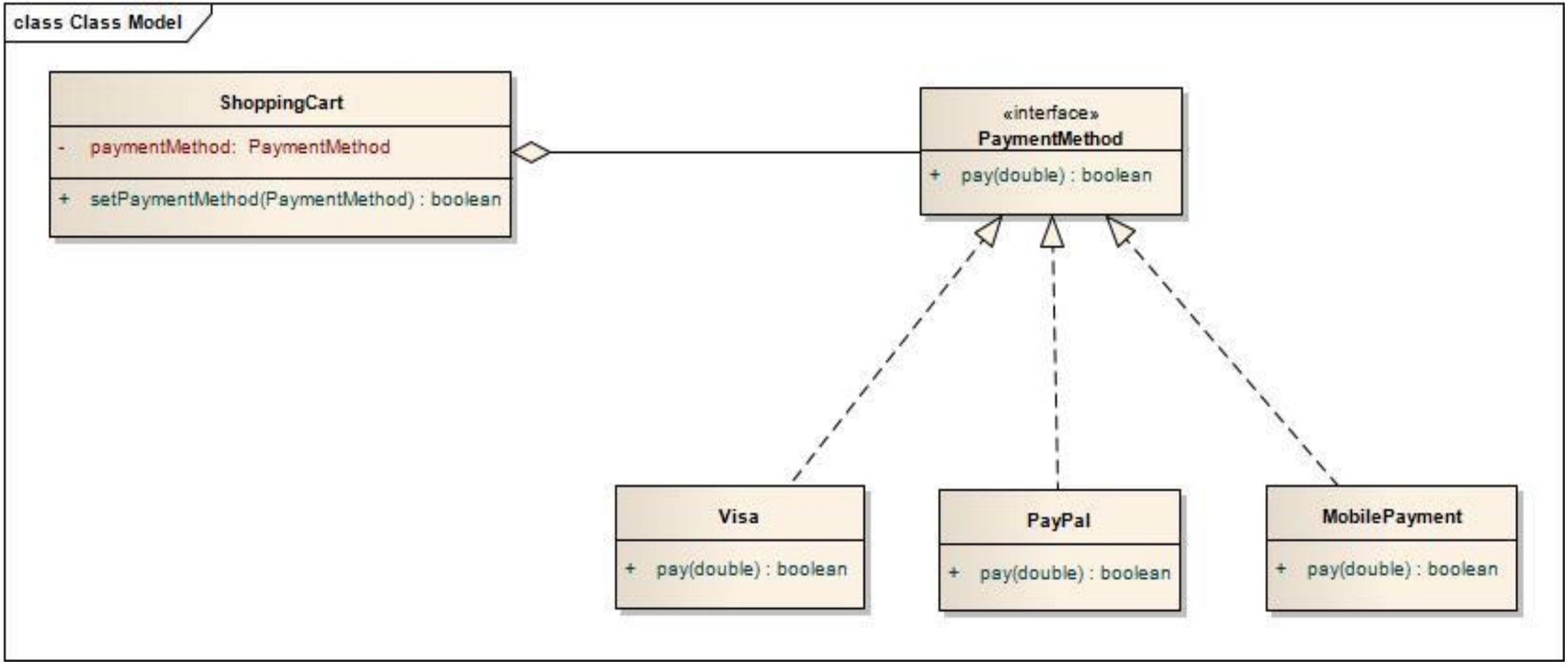
5. Setting behavior dynamically

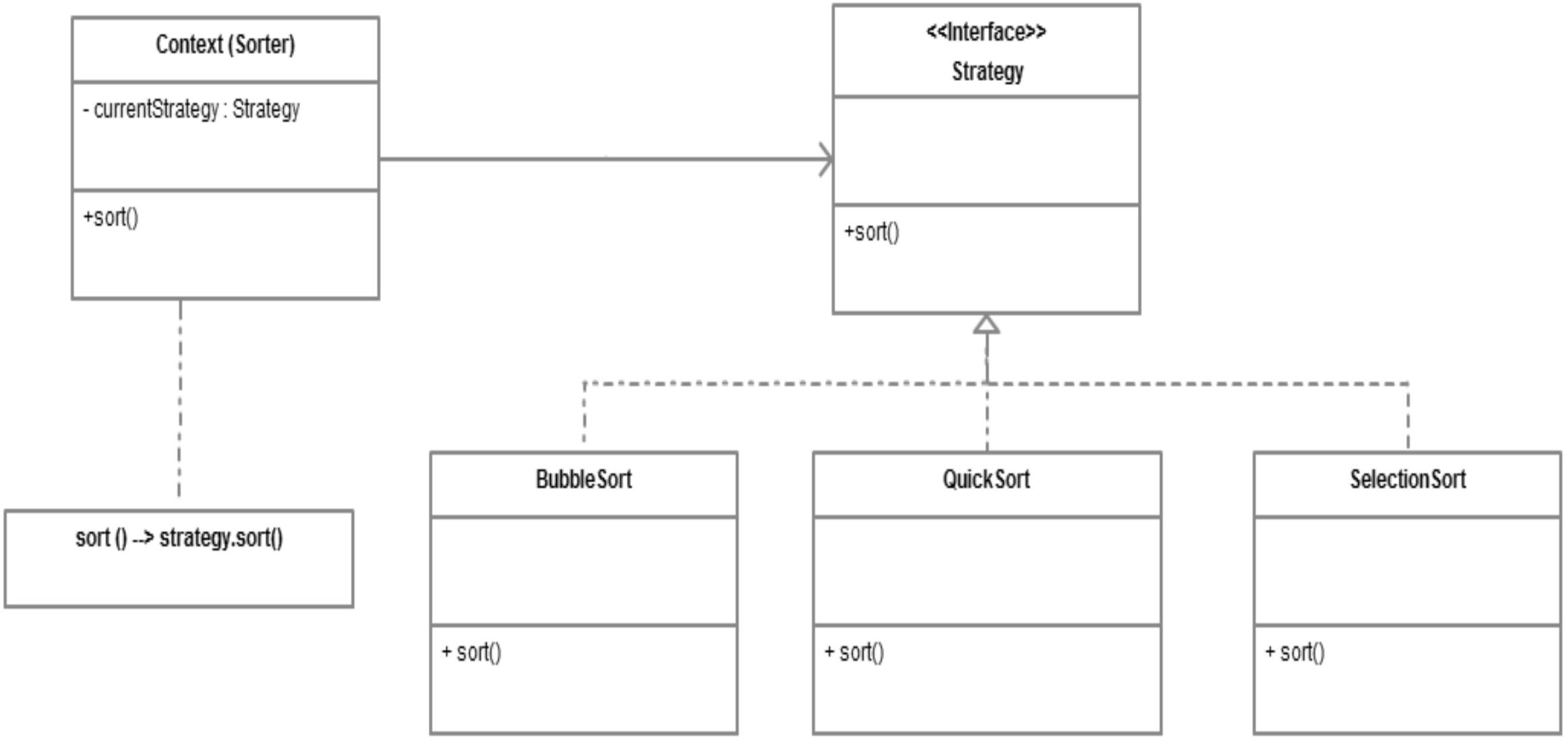
```
void MobilePhone::setFrontCameraBehavior(FrontCameraBehavior* fcb)
{
    frontCameraBehavior = fcb;
}

void main()
{
    MobilePhone* stone = new SamsungKeystone();
    stone->takeFrontCamera();
    stone->setFrontCameraBehavior(new FrontCamera());
    stone->takeFrontCamera();
}
```

```
I don’t have a front camera.
I’m taking a selfie.
```


<<interface>>
IHeartbeat
(from TopCoder.Heartbeat)
+<<get>>IsRunning() : bool
+<<get>>Message() : IMessage
+<<get>>HasException() : bool
+<<get>>NextException() : Exception
+<<get/set>>ExceptionQueueCapability() : int
+AddTrigger(trigger:ITrigger)
+RemoveTrigger(trigger:ITrigger) : bool
+RemoveAllTriggers() : IList
+ContainsTrigger(trigger:ITrigger) : bool
+Start()
+Stop()
AbstractHeartbeat
(from TopCoder.Heartbeat)
#isRunning : bool = false
#exceptionQueueCapability : int = 100
#triggers : IList = ArrayList.Synchronized(new ArrayList())
#message : IMessage = null
#exceptions : Queue = Queue.Synchronized(new Queue())
+<<get>>HasException : bool
+<<get>>NextException : Exception
+<<get>>IsRunning : bool
+<<get/set>>ExceptionQueueCapability : int
+<<get>>Message : IMessage
+AddTrigger(trigger:ITrigger)
+RemoveTrigger(trigger:ITrigger) : bool
+RemoveAllTriggers() : IList
+ContainsTrigger(trigger:ITrigger) : bool
+Start()
+Stop()
#OnHeartbeatEvent(sender:object, e:EventArgs)
+GetEnumerator() : IEnumerator
<<implementation>>
HttpHeartbeat
(from TopCoder.Heartbeat)
-webRequest : HttpWebRequest = null
+<<get>>Address : string
-ResponseCompleteCallback : AsyncCallback = new AsyncCallback(OnResponseComplete)
+HttpHeartbeat(address:string) : HttpHeartbeat
+HttpHeartbeat(msg:IMessage) : HttpHeartbeat
+HttpHeartbeat(uri:Uri) : HttpHeartbeat
+HttpHeartbeat(webRequest:HttpWebRequest) : HttpHeartbeat
#OnHeartbeatEvent(sender:object, e:EventArgs)
#OnResponseComplete(ar:IAsyncResult)
<<implementation>>
StreamHeartbeat
(from TopCoder.Heartbeat)
-stream : Stream = null
+<<get>>Stream : Stream
-SendCompleteCallback : AsyncCallback = new AsyncCallback(OnWriteComplete)
+StreamHeartbeat(stream:Stream, data:byte[]) : StreamHeartbeat
+StreamHeartbeat(stream:Stream, msg:IMessage) : StreamHeartbeat
+StreamHeartbeat(client:TcpClient, data:byte[]) : StreamHeartbeat
+StreamHeartbeat(client:TcpClient, msg:IMessage) : StreamHeartbeat
#OnHeartbeatEvent(sender:object, e:EventArgs)
#OnWriteComplete(ar:IAsyncResult)
0..*
<<handle>>
<<raise>>
<<delegate>>
HeartbeatEventHandler
(from TopCoder.Heartbeat)


class Class Model
ShoppingCart
- paymentMethod: PaymentMethod
+ setPaymentMethod(PaymentMethod) : boolean
«interface»
PaymentMethod
+ pay(double) : boolean
Visa
+ pay(double) : boolean
PayPal
+ pay(double) : boolean
MobilePayment
+ pay(double) : boolean


Context (Sorter)
- currentStrategy : Strategy
+sort()
<<Interface>>
Strategy
+sort()
sort () --> strategy.sort()
BubbleSort
+ sort()
QuickSort
+ sort()
SelectionSort
+ sort()

PROS AND CONS

1. Advantages

- Reduces long lists of conditionals.
- Avoid duplicate code. Keep class changes from forcing other class changes (reuse/interchange algorithms/behaviors) depends on the context.
- Encapsulating the algorithm make it easier to switch, understand and extend.
- Encapsulate/hide complicate/secret code from users.
- The client can choose among strategies with different implementations.

2. Disadvantages

- Increased number of objects/classes.
- Change the structure of program easily to confuse if you don't know the structure well.
- A client must understand how Strategies differ before it can select the appropriate one.
- Communication overhead between Strategy and Context. There will be times when the context creates and initializes parameters that never get used.

REFERENCE

- Head first design pattern - Elisabeth Freeman Eric Freeman.
- Design Patterns - The "Gang of Four"
- http://www.bogotobogo.com/DesignPatterns/strategy.php
- https://vi.wikipedia.org/wiki/Strategy_pattern
- https://www.codeproject.com/articles/1018930/strategy-design-pattern-explained-with-a-real-worl
- https://prathapgivantha.wordpress.com/2012/09/14/strategy-pattern/
- http://www.java67.com/2014/12/strategy-pattern-in-java-with-example.html#ixzz4Sj93o0qz