OODP– Session 9

Session times

PT – Thursday 18:00-21:00

room: School of Pharmacy, John Hanbury Lecture Theatre

FT - Tuesday 13:30-17:00 room: Malet 404

Email: oded@dcs.bbk.ac.uk

Web Page: http://www.dcs.bbk.ac.uk/~oded

Visiting Hours: Tuesday 17:00 to 19:00

Chain of Responsibility Design pattern

Why do we need this pattern?

We will start by trying to do without it.

Clone the project from

git://github.com/odedlac/Bank.git

This project contains raw code for automating a bank teller.

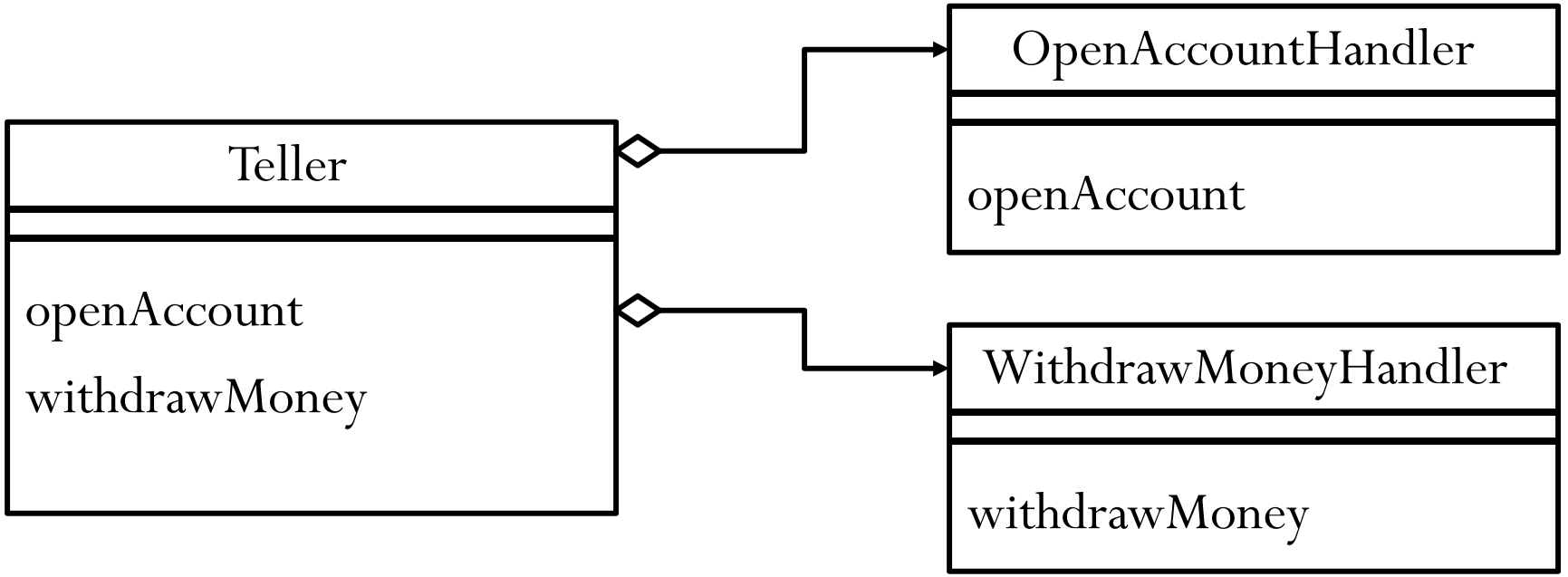
Bank Project

The project contains raw code for automating a bank teller

At this stage the teller can do only two things

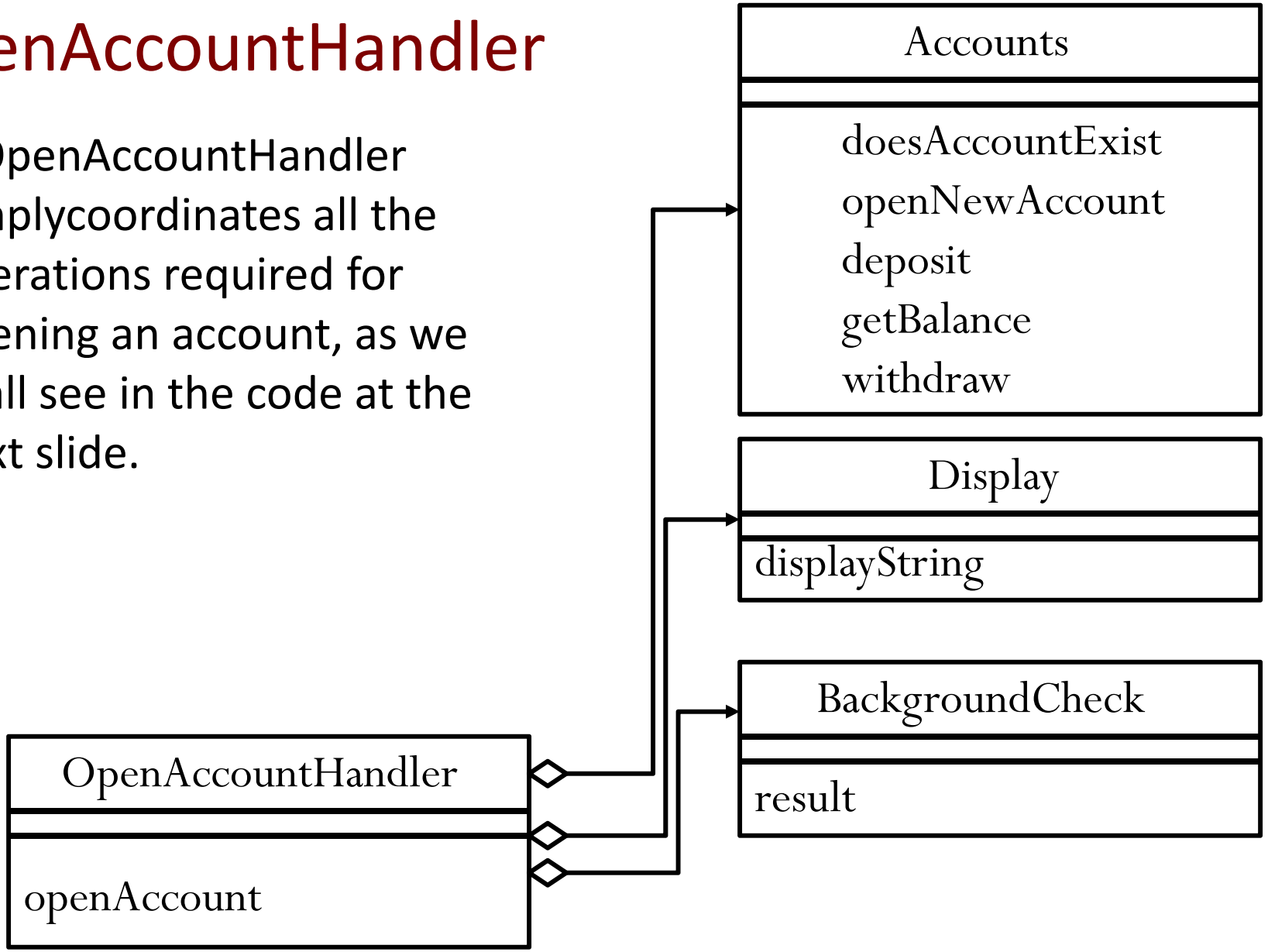
1. Open an account
2. Withdraw money

Technically what teller does is delegate the actual doing of the operations to two other classes.



OpenAccountHandler

The OpenAccountHandler simplycoordinates all the operations required for opening an account, as we shall see in the code at the next slide.



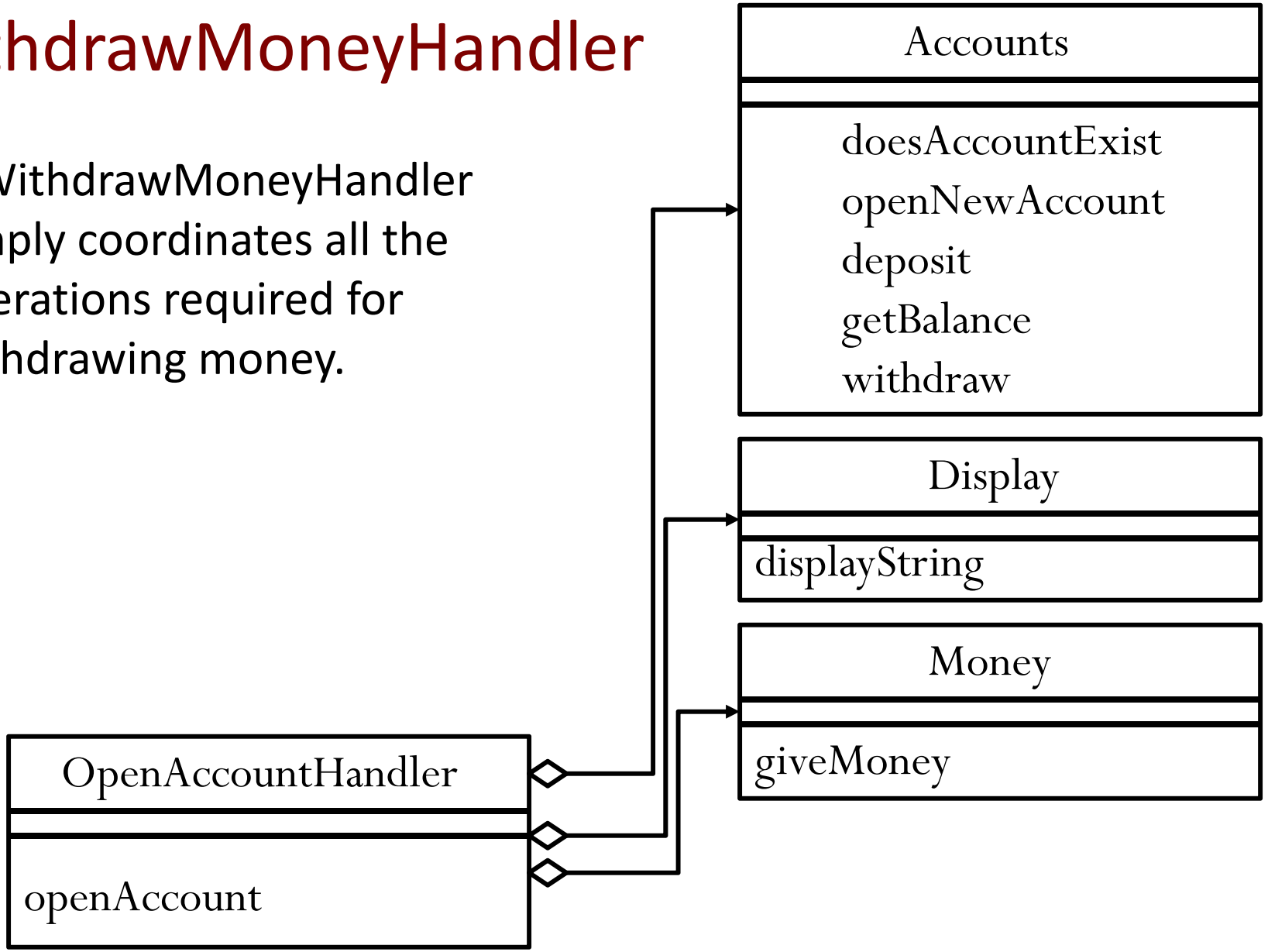
```
public Boolean openAccount(String string) {
        if(!backGroundCheck.result(string)){
                display.displayString("Sorry request refused");
                return false;
        }
        Integer accountNumber = accounts.openNewAccount();
        display.displayString("Request accepted. New account number is: " +
accountNumber);
        return true;
}
```

Background test hasn't failed. Account opened and customer informed through display.

Runs a background check if it fails then it returns false since.

WithdrawMoneyHandler

The WithdrawMoneyHandler simply coordinates all the operations required for withdrawing money.

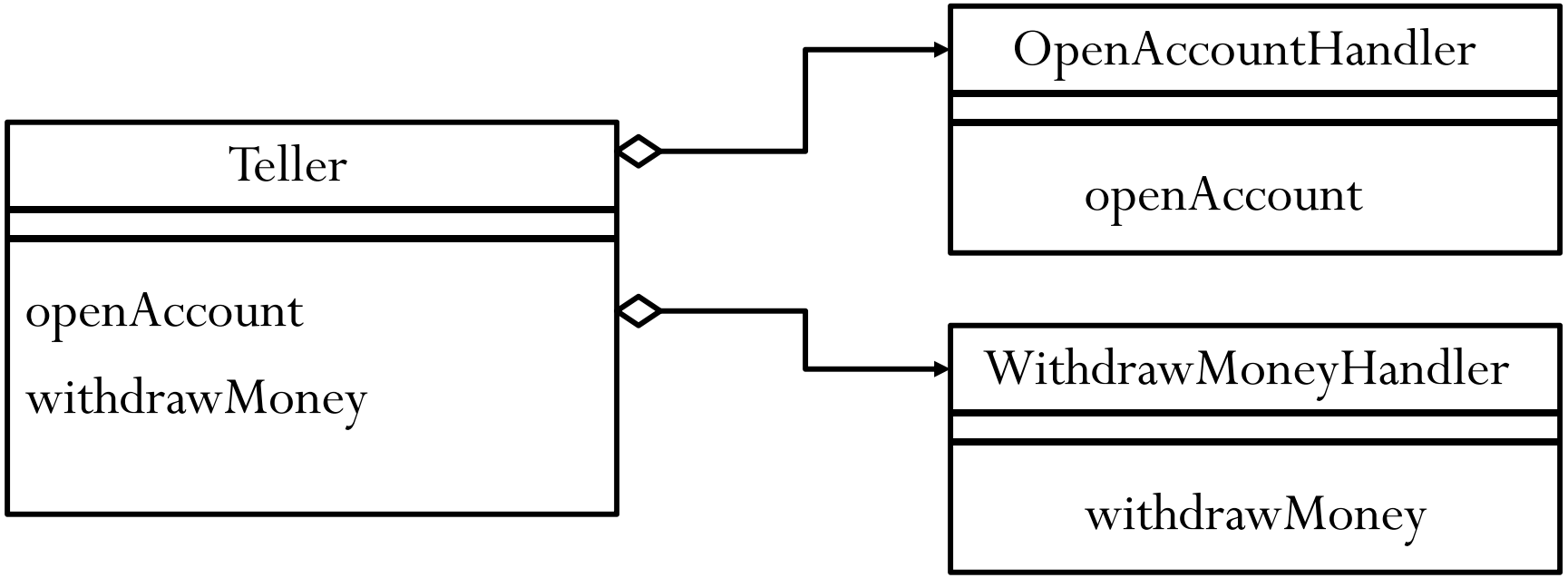


Bank Project

Why would we need such a solution.

One possible scenario:

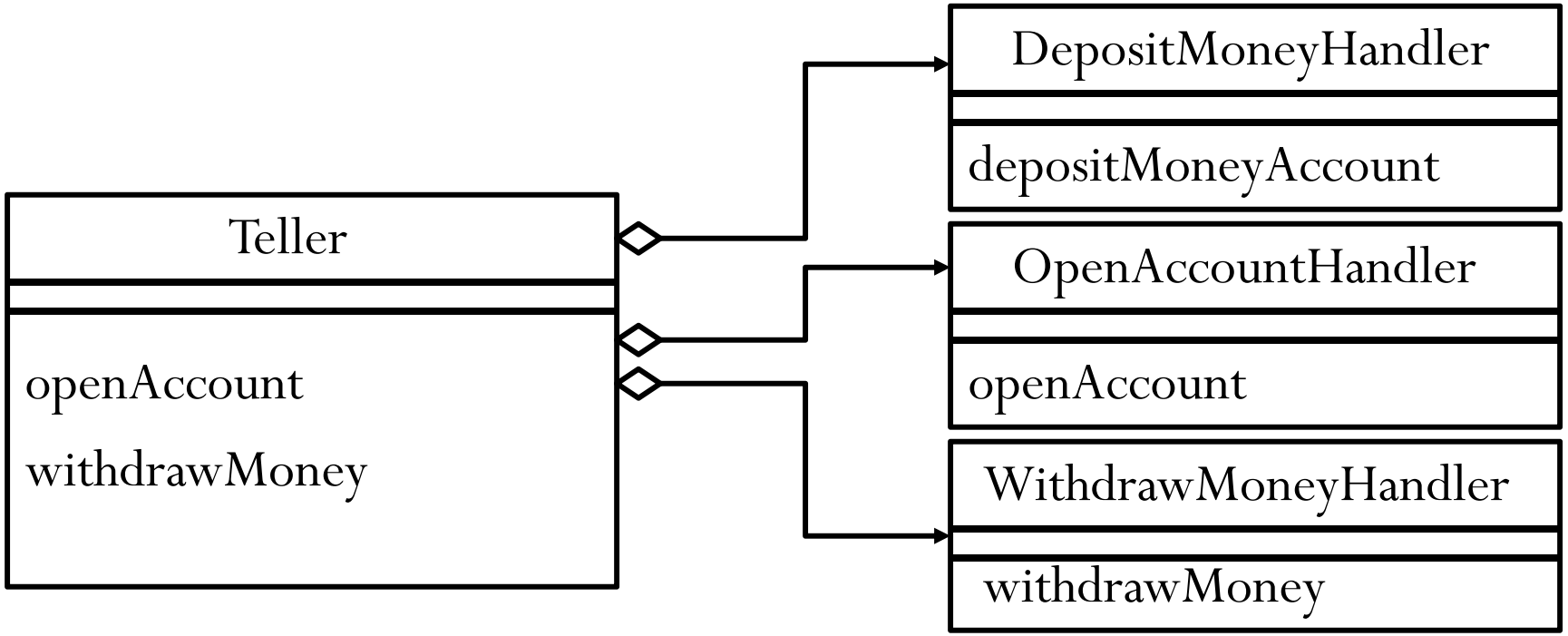
- We created the teller class because of the controller grasp pattern
- But after some agile iterations it turns out that the variety and number of requests the teller has to handle is causing us Cohesion issues.
- Our solution delegate



What is wrong with this solution

In order to understand what is wrong with this solution lets add another possible handler to the teller.

You have 15 minutes to write a fully supported depositMoneyHandler. It should be implemented on the same lines as the other Handlers.



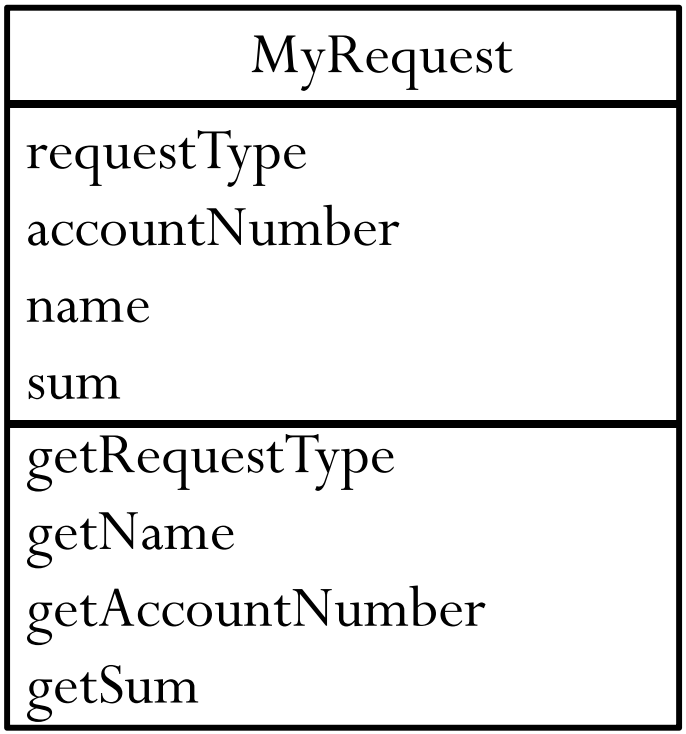
Are you happy with this solution?

Coupling – we had to add methods to ‘Teller’ and we don’t want to do that because the number of such methods may be very large.

Any idea how to solve this?

Idea

1. Create a request class that contains all the relevant information.
2. Use a ‘myRequest’ object in order to tell the ‘teller’ object what to do
3. The ‘teller’ object will delegate it to each of the “requestHandlers”
4. A request handler should check whether he can handle the request and only then handle it



Clone code from
git://github.com/odedlac/BankWithRequests.git

Comparing new to old code (test1 in TellerTest)

Old:

```
public void test1() {
        assertFalse("Opened account when shouldn't", classUnderTest.openAccount("Oded"));
}
```

New:

```
public void test1() {
        MyRequest myRequest = new MyRequest("OpenAccount","Oded",0,0);
        assertFalse("Opened account when shouldn't",
        classUnderTest.handleRequest(myRequest));
}
```

Adding 'MyRequest' made dealing with the teller a bit more complicated

Comparing new to old code (changes to teller)

Old:

```
public boolean openAccount(String string) {
            return openAccountHandler.openAccount(string);
}

public boolean withdraw(Integer accountTested, Integer sumTested) {
            return withdrawMoneyHandler.withdraw(accountTested, sumTested);
}
```

New:

```
public boolean handleRequest(MyRequest myRequest) {
            if(openAccountHandler.handleRequest(myRequest)){
                        return true;
            }
            if(withdrawMoneyHandler.handleRequest(myRequest)){
                        return true;
            }
            return false;
}
```

Adding ‘MyRequest’ reduced number of methods in teller, and simplified the delegation

Comparing new to old code (changes to withdrawMoneyHandler)

Old:

```
public boolean withdraw(Integer accountNumber, Integer sum) {…}
```

New:

```
private boolean withdraw(Integer accountNumber, Integer sum) {…}

public boolean handleRequest(MyRequest myRequest) {
// TODO Auto-generated method stub
          if(myRequest.getRequestType().equals("WithdrawMoney")){
                    return withdraw(myRequest.getAccountNumber(),myRequest.getSum());
          }
          return false;
}
```

Adding forced adding a new method to handle ‘myRequest’ (the withdraw method is now private)

Challenge

You have 12 minutes to write a fully supported depositMoneyHandler. It should be implemented on the same lines as the other Handlers.

Are you happy with this solution?

Coupling – we had to add code in‘Teller’ and we don’t want to do that because the method may grow very large.

Any idea how to solve this?

Lets apply The Chain of Responsibility Design Pattern

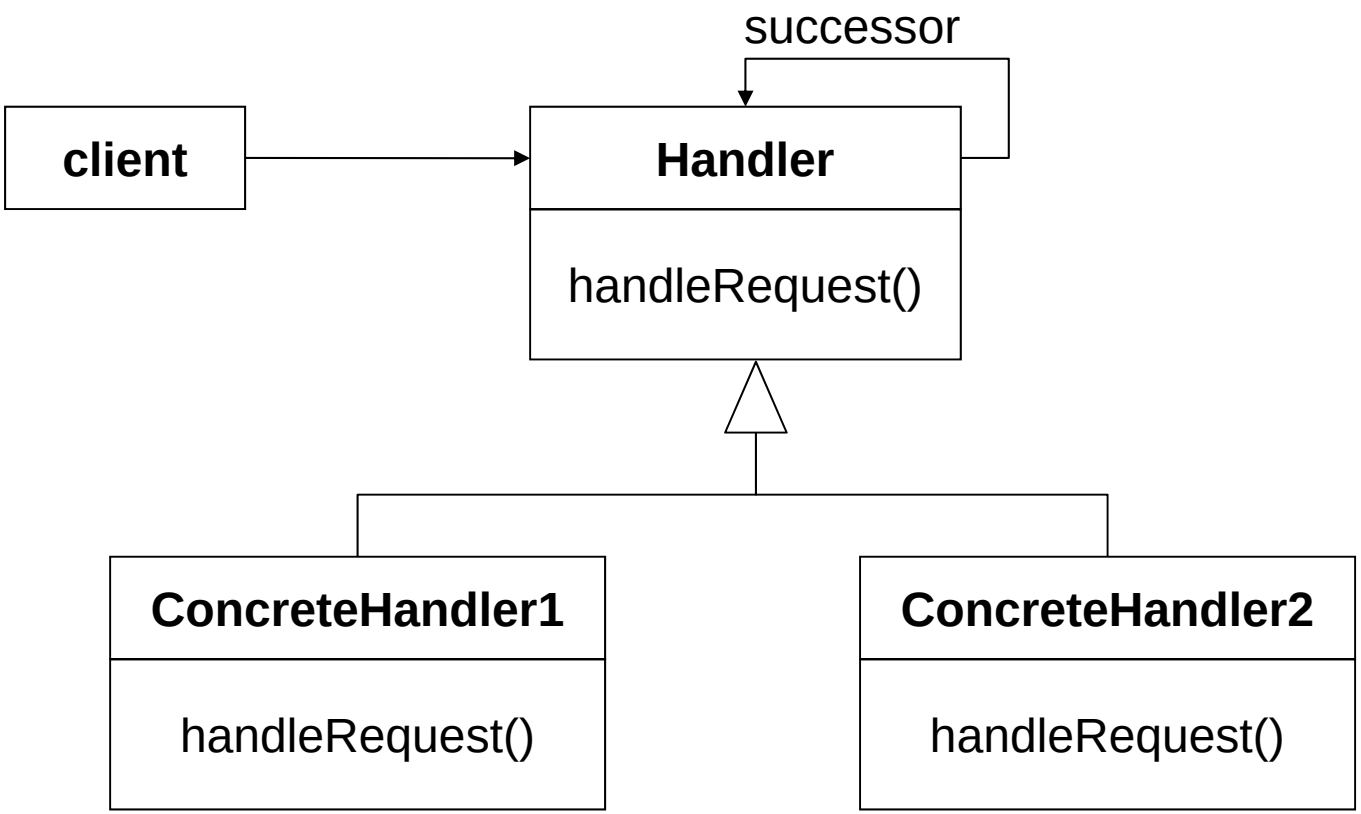
Chain of Responsibility

Problem: How to handle a request when it is not known which object should deal with it?

Motivation:

- In the example (with the GRASP pattern) we had a controller.
- What if in order to avoid cohesion we need many controllers, one for each complex operation
- On top of that we want to be able to add more operations

Chain of Responsibility



Pattern implementation

1. Clone the project from

   git://github.com/odedlac/BankWCOR.git

Handler Code

```
public abstract class Handler {

        protected Handler handler;

        public Handler() {
                this.handler = null;
        }

        public void setHandler(Handler handler){
                this.handler = handler;
        }

        public abstract Boolean handleRequest(MyRequest
myRequest);

}
```

Straight forward implemntation – 'handler' is a reference to the next in line. By default it is nul.

'handleRequest' method must be implemented since it is abstract

Comparing newer to new code

New:
TellerTest

```
public void setUp() throws Exception {
            Accounts accounts = new Accounts();
            accounts.deposit(accounts.openNewAccount(), 100);
            accounts.deposit(accounts.openNewAccount(), 200);

            Money money = new Money();
            Display display = new Display();
            BackgroundCheck backGroundCheck  = new BackgroundCheck();

            WithdrawMoneyHandler withdrawMoneyRequest = new
WithdrawMoneyHandler(accounts,money,display);
            OpenAccountHandler openAccountRequest = new
OpenAccountHandler(accounts,backGroundCheck,display);

            classUnderTest = new
Teller(openAccountRequest,withdrawMoneyRequest);
}
```

Newer:
ChainTest

```
public void setUp() throws Exception {
            Accounts accounts = new Accounts();
            accounts.deposit(accounts.openNewAccount(), 100);
            accounts.deposit(accounts.openNewAccount(), 200);

            Money money = new Money();
            Display display = new Display();
            BackgroundCheck backGroundCheck  = new BackgroundCheck();

            Handler withdrawMoneyRequest = new WithdrawMoneyHandler(accounts,money,display);
            classUnderTest = new OpenAccountHandler(accounts,backGroundCheck,display);
            classUnderTest.setHandler(withdrawMoneyRequest);
}
```


Pattern made ‘Teller’ redundant. Creation of Handlers is slightly harder.

Comparing newer to new code (WithdrawMoneyHandler)

New:

```
public boolean handleRequest(MyRequest myRequest) {
// TODO Auto-generated method stub
            if(myRequest.getRequestType().equals("WithdrawMoney")){
                        return withdraw(myRequest.getAccountNumber(),myRequest.getSum());
            }
            return false;
}
```

Newer:

```
public Boolean handleRequest(MyRequest myRequest) {
            if(myRequest.getRequestType().equals("WithdrawMoney")){
                        return withdraw(myRequest.getAccountNumber(),myRequest.getSum());
            } else if(super.handler == null){
                        return false;
            }
            return super.handler.handleRequest(myRequest);
}
```

Request handler slightly more complicated because of chain requirements

Pattern implementation

1. You have 10 minutes to write a fully supported depositMoneyHandler. It should be implemented on the same lines as the other Handlers.

2. Any idea of which other pattern maybe nice to have here.

3. If you have time to spare then maybe try to implement it

Chain of Responsibility, Pros and Cons

Pro

1. Reduced coupling
2. Flexibility – easy to add new handlers

Cons

1. Need ensure the chain is created properly
2. Requests might not be handled
3. The request object may become too large

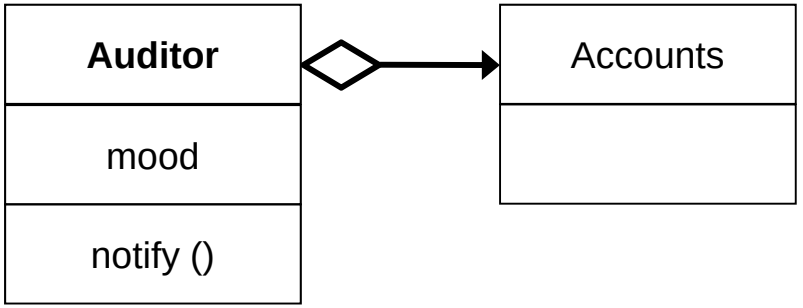
Observer

Auditor

- We want to simulate an auditor that wants to observe every single change in each account
- How would you design such a class

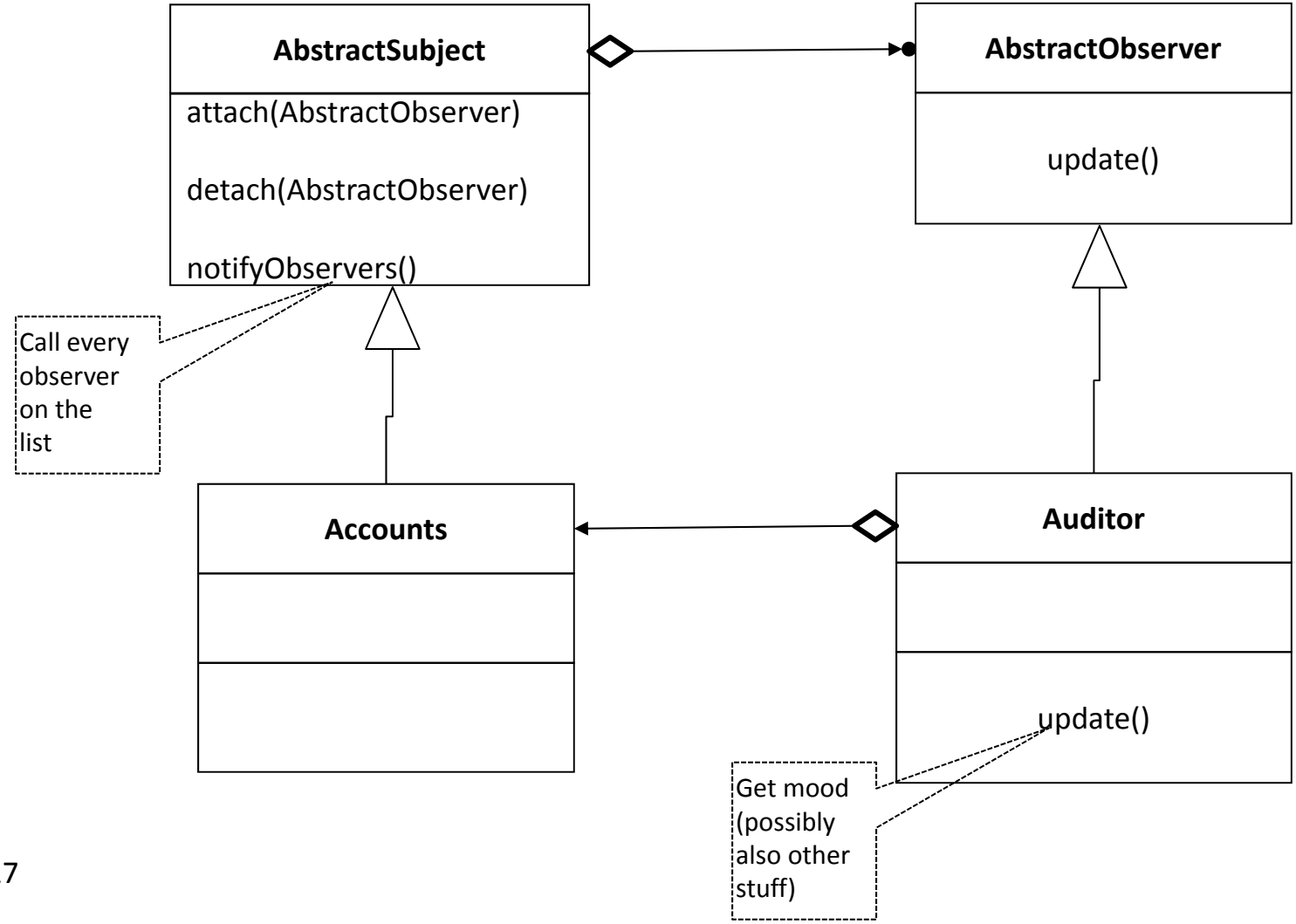
What is the problem with this solution?

What if there is more than one auditor and they may want to come and go?



Auditor is an Observer

- Solution: “teach them the subject observer relation”



Observer Implementation

1. Clone the project from

   git://github.com/odedlac/BankWithAuditors.git

AbstractSubject Code

```
public class AbstractSubject {

            private Set<AbstractObserver> observers;

            protected AbstractSubject(){
                        observers = new HashSet<AbstractObserver>();
            }

            public void attach(AbstractObserver abstractObserver){
                        observers.add(abstractObserver);
            }
            public void detach(AbstractObserver abstractObserver){
                        observers.remove(abstractObserver);
            }

            protected void notifyObservers(){
                        java.util.Iterator<AbstractObserver> auditorsIterator =
            observers.iterator();
                        while(auditorsIterator.hasNext()) {
                                    auditorsIterator.next().update();
                        }
            }
}
```

AbstractObserver Code

```
public abstract class AbstractObserver {
            abstract void update();
}
```

Auditor Code

```
public class Auditor extends AbstractObserver {

	private Accounts accounts;
	private Integer number;

	public Auditor(Accounts accounts,Integer number) {
		this.accounts = accounts;
		this.number = number;
	}
/* (non-Javadoc)
 * @see code.AbstractObserver#update()
 */
	@Override
	void update() {
		System.out.println(number + " - Someone is messing with the
accounts  " + accounts.doesAccountExist(1));
	}

}
```

Changes to Accounts code

```
public class Accounts extends AbstractSubject {…

        public Integer openNewAccount() {
                super.notifyObservers();
                Integer accountNumber = numberToBalance.size()+1;
                numberToBalance.put(accountNumber, 0);
                return accountNumber;
        }
```

Auditors test code

```
public class AuditorTest {

        Accounts accounts;
        AbstractObserver auditor1;
        AbstractObserver auditor2;

        @Before
        public  void setUp() throws Exception {
                accounts = new Accounts();
                accounts.openNewAccount();
                accounts.openNewAccount();

                auditor1 = new Auditor(accounts,1);
                auditor2 = new Auditor(accounts,2);

                accounts.attach(auditor1);
                accounts.attach(auditor2);

        }
```

Pattern implementation

1. You have 15 minutes to
   A. Add an observer to some other class
   B. Try to make the whole thing fail because of observer design pattern abuse

Discussion

Why Use?

- A change to one object requires change to another object
- One object can notify others without any dependency on their class

Issues

- Need some mechanism to say what changed
- Without maintenance, may get an unwanted cascade of updates

More

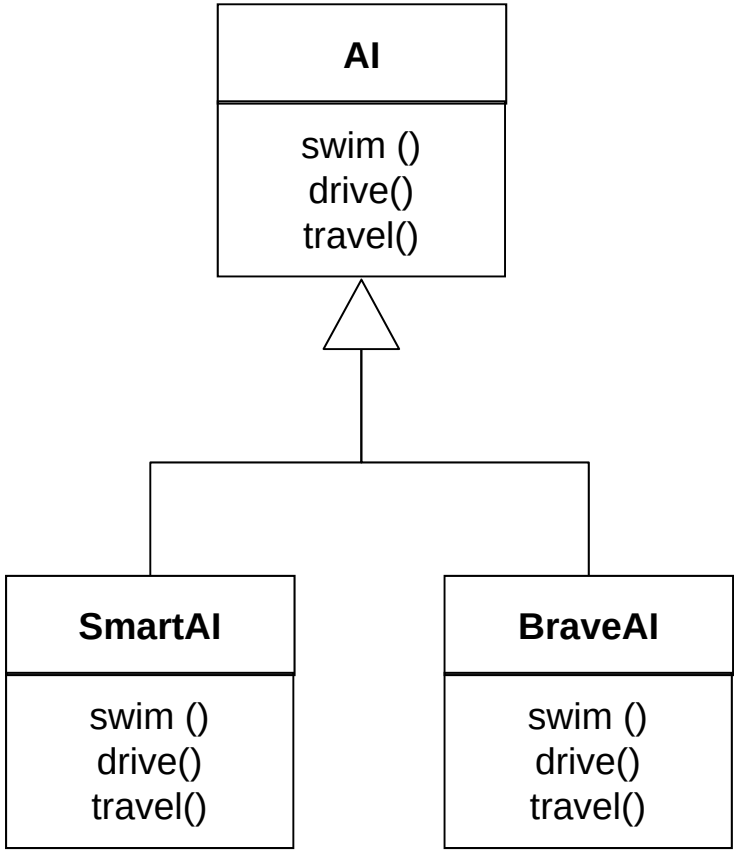
Advanced Patterns

Bridge

DriverAI

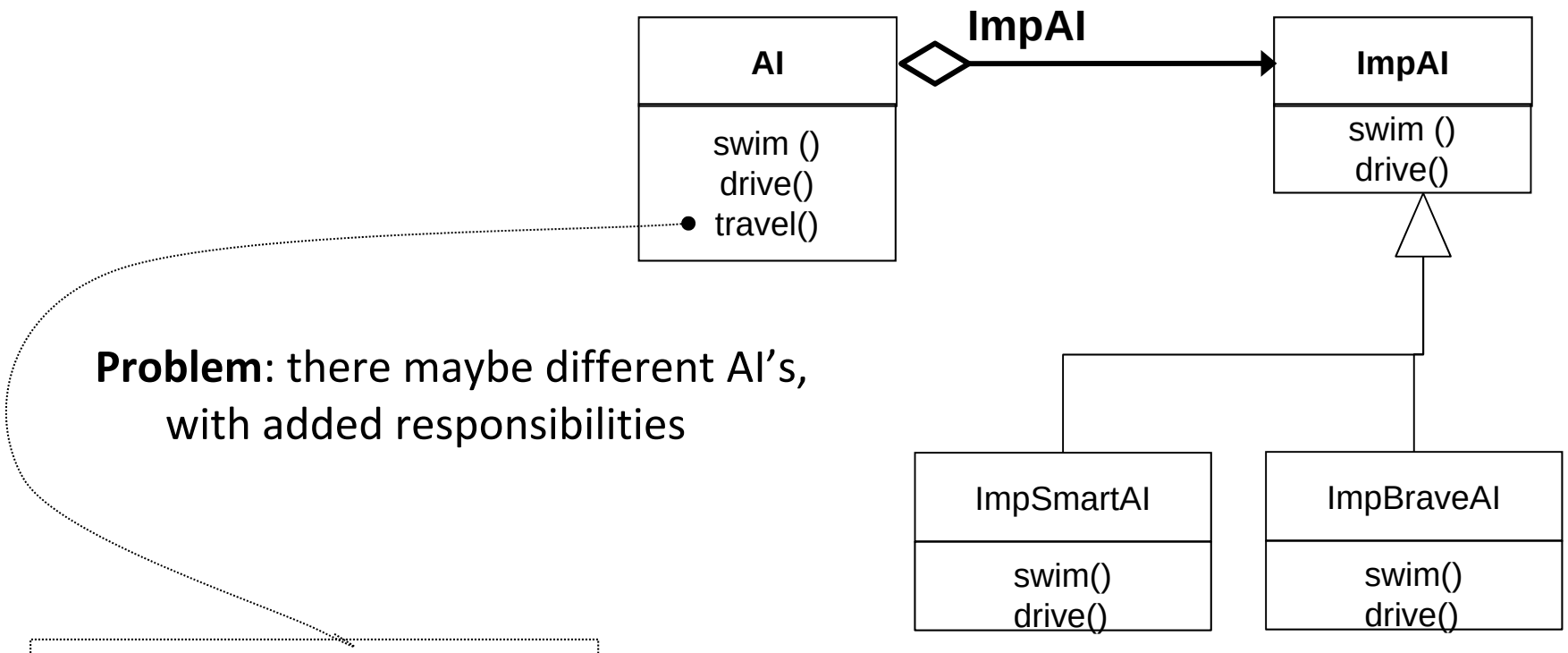
Problem: We want to decouple an abstraction from its implementation

Why can't we settle for inheritance?



Bridge Pattern

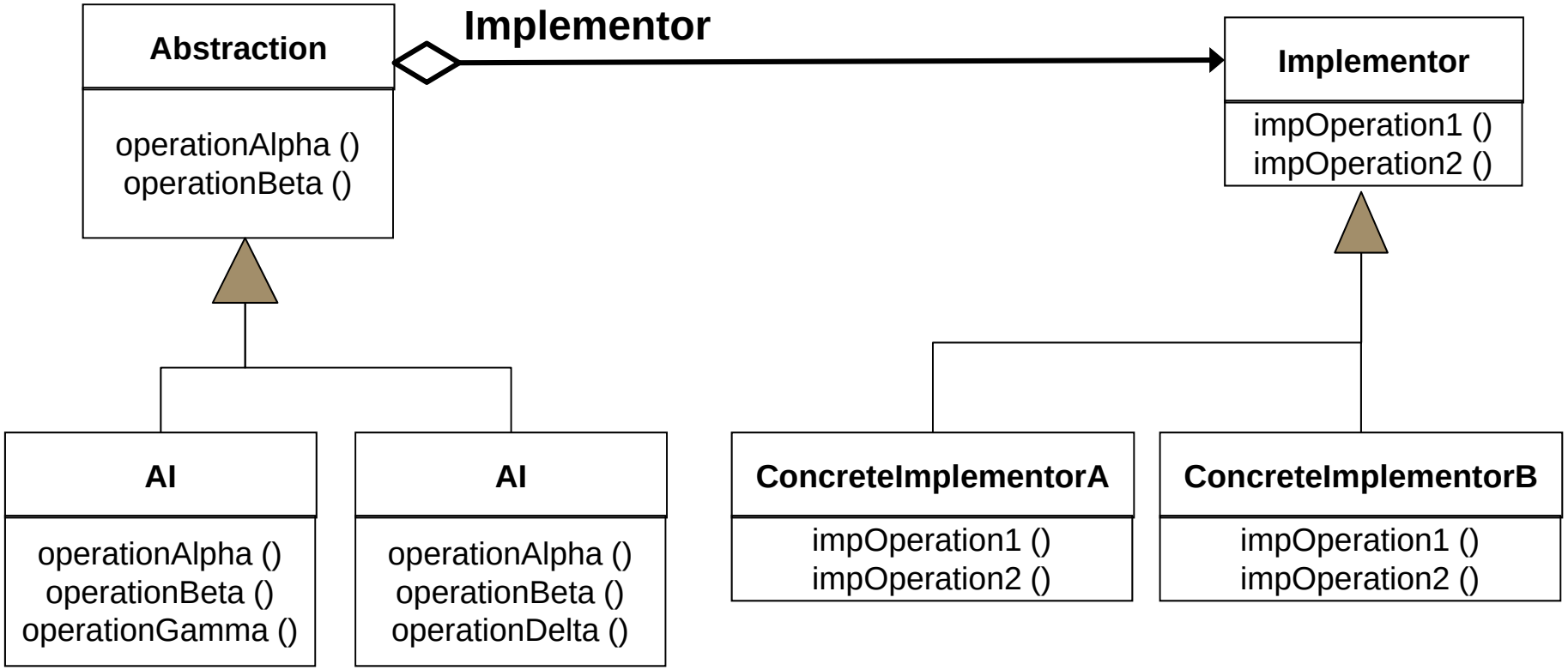
Solution: delegate implementation



Problem: there maybe different AI's, with added responsibilities

```
void AI::travel() {
        ImpAI->drive();
        ImpAI->swim();
}
```

Solution: Inheritance



Discussion

Why use?

- Avoiding permanent binding
- Abstraction and implementations are extensible by binding
- Implementation change have no impact on clients
- Implementation hidden
- Implementations can be shared

(what is the difference from strategy)

Something to do at home

1. Do the first example from scratch with mock objects for classes such as ‘Accounts’ etc.
2. Use a ‘MyRequest’ mock object in the second example and write tests to check how it was treated
3. Like two but see if you can learn how to spy
4. Draw class UML diagrams for the examples
5. Apply other patterns to the examples (if you can find a good place to do so)