THE OB SERVER
AKA
"THE INFORMANT"

When Not Knowing Can Get You Taken For A Ride…

Rich J Miller
CSPP 51023
Winter 2011

WHAT IS THE OBSERVER PATTERN ?

Generally speaking…

the observer pattern defines a behavioral ONE to MANY relationship in that one object contains some state information and notifies several other objects when that state has been changed…

WHY USE THE OBSERVER PATTERN ?

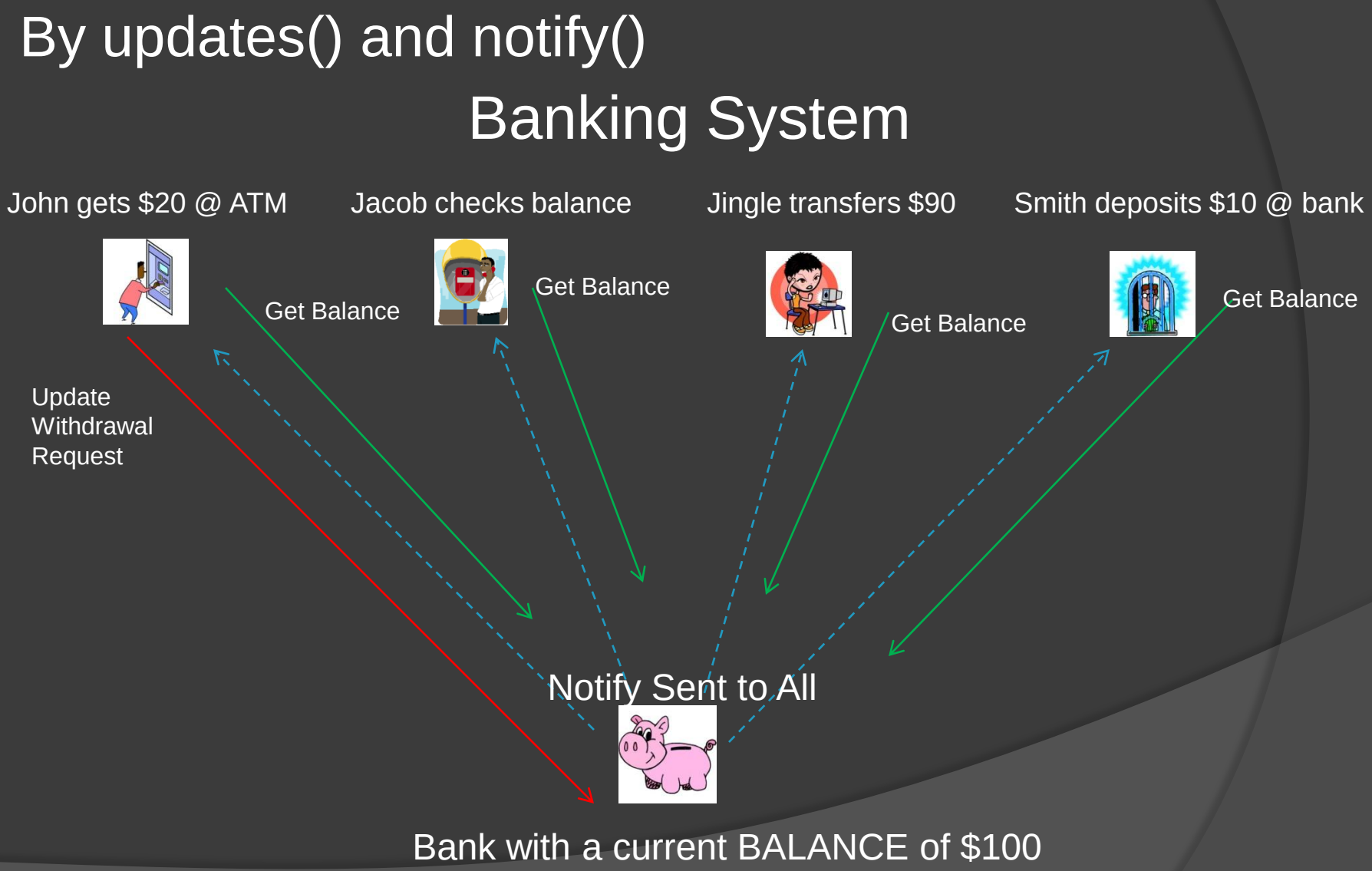
Moreover…

when there are several objects we will call observers that are all dependant on a single source of data, each observer acting on this data in various ways when the state of the data is changed. This data can be encapsulated into a single object called a subject.

Allowing each observer to operate “independently of the other while remaining synchronized with the state of the data.

HOW IS IT POSSIBLE ?

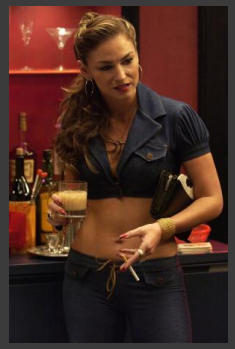
By updates() and notify()



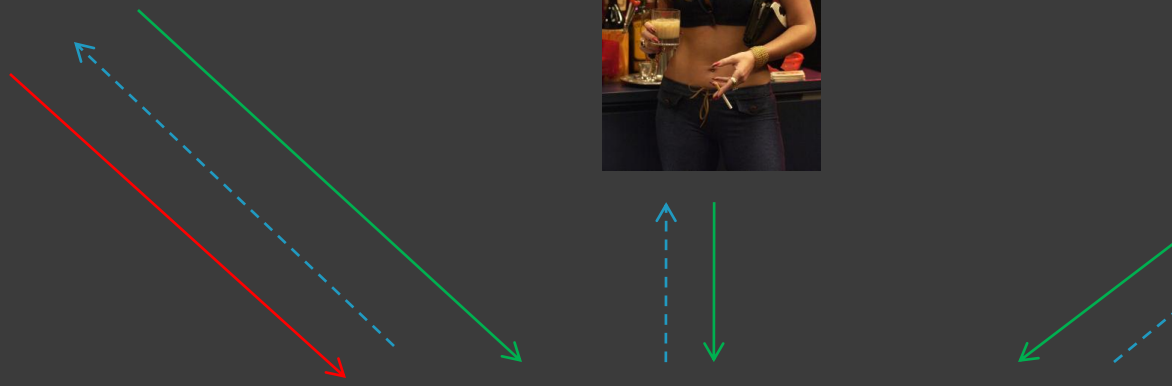
THE INFORMANTS

Big “P…”

Adrianna



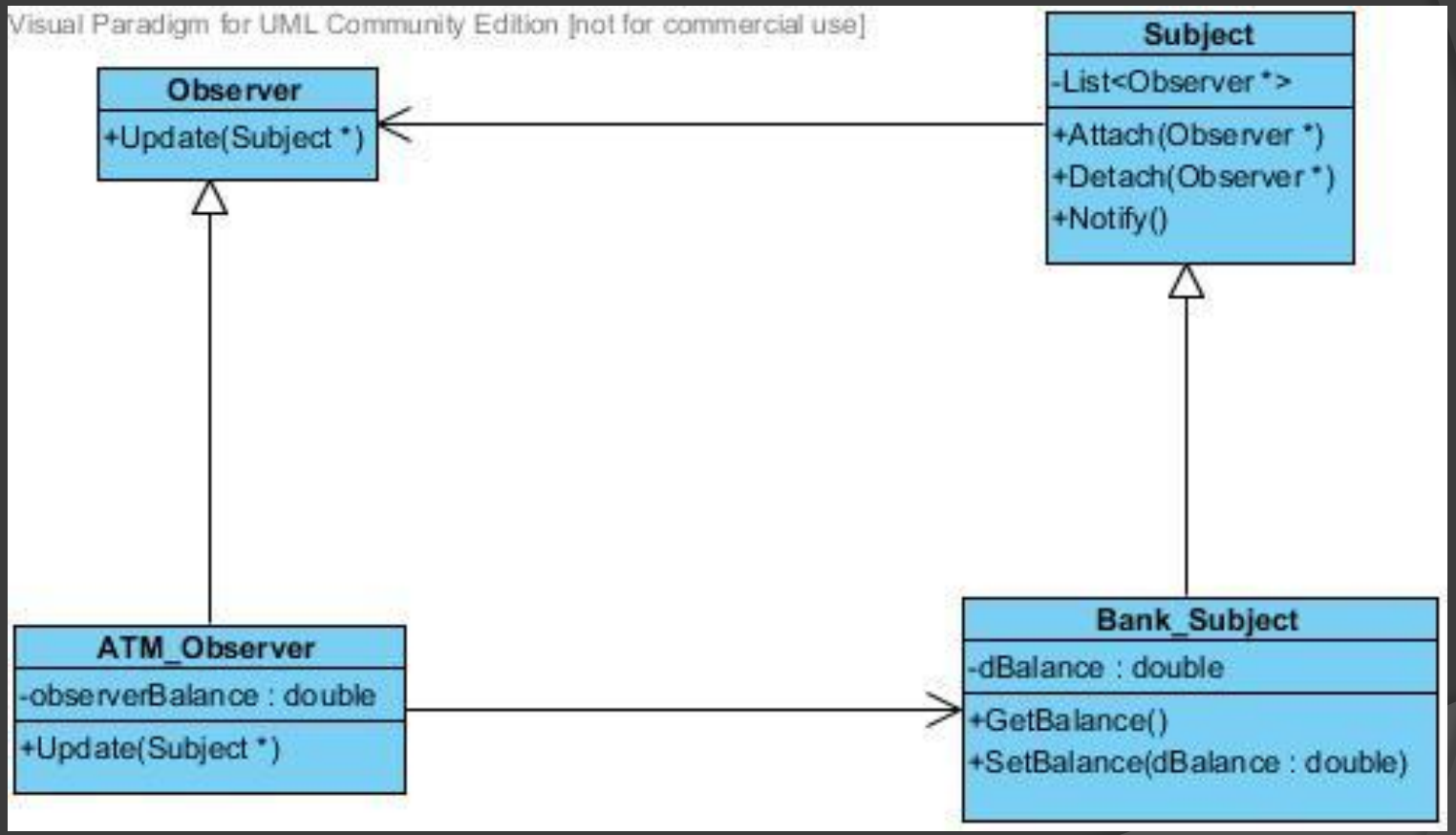
Massarone



Subject

State Data

U M L


Visual Paradigm for UML Community Edition [not for commercial use]
Observer
+Update(Subject *)
Subject
-List<Observer *>
+Attach(Observer *)
+Detach(Observer *)
+Notify()
ATM_Observer
-observerBalance : double
+Update(Subject *)
Bank_Subject
-dBalance : double
+GetBalance()
+SetBalance(dBalance : double)

THE CODE

```
class Observer
{
  public:
    virtual ~Observer();
    virtual void Update(Subject* bankSubject) = 0;
  protected:
    Observer();
};

class Subject
{
  public:
    virtual ~Subject();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();

  protected:
    Subject();

  private:
    list<Observer*> *observerList;
};

void Subject::Attach (Observer* obsAdd)
{
  observerList->push_back(obsAdd);
}
```

```
void Subject::Detach (Observer* obsRemove)
{
  observerList->remove(obsRemove);
}

void Subject::Notify ()
{
  list<Observer*>::iterator i;

  for(i = observerList.begin();
      i != observerList.end();
      i++)
  {
    ((Observer*)i)->Update(this);
  }
}

class BankSubject : public Subject
{
  public:
    BankSubject();

    virtual double GetBalance();
    virtual void SetBalance(double);

  private:
    double dMainBalance;
}
```

THE CODE

```
void BankSubject::SetBalance(double d)
{
   dMainBalance = d;
   notify();
}

double BankSubject::GetBalance()
{
   return dMainBalance;
}

class ATM_Observer : public Observer
{
   public:
      ATM_Observer(BankSubject*);
      virtual ~ATM_Observer();

      virtual void Update(Subject*);
      void ChangeBalance(double newBalance);

   private:
      BankSubject* thisBankSubject;
      double myBalance;
}

ATM_Observer::ATM_Observer(BankSubject* bs)
{
   thisBankSubject = bs;
   thisBankSubject->Attach(this);
}
```

```
ATM_Observer::~ATM_Observer()
{
   thisBankSubject->Detach(this);
}

void ATM_Observer::Update(BankSubject* bs)
{
   if(thisBankSubject == bs)
      myBalance == thisBankSubject->GetBalance();
}

void ATM_Observer::ChangeBalance(double d)
{
   thisBankSubject->SetBalance(d)
}
```