UNIVERSITY OF Hull

Storing Data in Objects

Rob Miles

Department of Computer Science

28d 08120 Programming 2

Objects and Items

- I have said for some time that you use objects to represent things in your problem
- Objects equate to the *nouns* in a description of a system
  - *"The Bank will contain a number of different Accounts"*
- Each of these will hold a lump of data which is important to your system
- The job of the programmer is to decide what goes into the objects and the things they need to do

Business Objects

- Objects which represent fundamental entities in the system that you are representing
  - They are sometimes called *Domain Objects*
- Four business objects, and one that isn't:
  - Customer
  - Receipt
  - Login Window
  - Tree
  - Photograph

Designing With Objects

- A Software Engineer will represent entities in the system with software objects
- The object holds data and it also does things for us
- At this level the design of a system is performed by deciding what objects are required, what data they must store and what they need to do for us
- This is really an extension of the “metadata driven” approach that we started with

Data In Objects

- Objects can contain data
- We can protect this by making it *private*
- If we make data private we need to provide public methods that allow the data to be used
- What the data is, and what you can do with it depends on the application you are building

Friendly Bank Requirements

- pay money into the account
- draw money out of the account
- find the balance
- change the name and address of the account holder
- get the name of an account holder
- get the address of an account holder
- change the overdraft limit on an account
- find the overdraft limit on an account

Account Class Data

- The Account class will need to hold data
- We have to go back to the requirements to decide what data is to be held
  - Also need to determine the type of the data and how we are going to represent it
  - Also need to consider data validation
- All this means the return of the *metadata*

An Account Class

```
class Account
{
    public string AccountName;
    public string AccountNumber;
    public decimal AccountBalance;
}
```

- This is our first attempt at an Account business object
- It only contains part of the system information
  - Other data fields will be required to complete the system
- We can create an array of these for the bank

Protecting Data in Classes

```
Account Rob = new Account ();
Rob.AccountBalance = 1000000;
```

- When we design our objects we need to consider how the data in them is going to be protected
- We want to avoid naughty programmers being able to make changes which would upset the state of our objects
- It is important that we control access to the variable that holds the `accountBalance`

An Account Class

```
class Account
{
    private string accountName;
    private string accountNumber;
    private decimal accountBalance;
}
```

- The data fields in the `Account` class have now been made private
- This means that code which is not part of the `Account` class can't change these values
- This protection is enforced at compile time

Protecting Data in Classes

```
Account Rob = new Account ();
Rob.accountBalance = 1000000;
```

- When the `accountBalance` field is made `private` it is impossible for code outside the `Account` class to access that field
- The above code will not compile, unless the statements are part of a method inside the `Account` object
- This is how data inside an object is protected

Using Private Data Fields

- If a data field is private this means that only code running inside the object can access it
- This might make you think that it is impossible for code outside the object to make use of the data in that object
- However, this is not the case, as the creator of the object can provide methods that will allow external use of these fields
- There are two kinds of methods that you can create which are called *accessors* and *mutators*

Accessors and Mutators

- An *accessor* method provides read access to a data field inside the object (sometimes called a *get* method)
  - Accessor methods return a value of some kind
- A *mutator* method allows you to change (mutate) a data field (sometimes called a *set* method)
  - The mutator can be given a value that will be used to change the field
  - It should make sure that the change is sensible
  - Mutator methods return a result that indicates whether they worked or not

Accessors and Mutators for the Account Balance

- We can write some of these methods for the balance value of an account
- There are three things the system will need to do with the balance value
  - Pay in funds
  - Withdraw funds
  - Find out the account balance
- Which are accessors and mutators?

Accessors and Mutators for the Account Balance

- We can write some of these methods for the balance value of an account
- There are three things the system will need to do with the balance value
  - Pay in funds
  - Withdraw funds
  - Find out the account balance
- Which are accessors and mutators?

A PayInFunds Method

```
class Account
{
    private decimal accountBalance;

    public void PayInFunds (decimal amount)
    {
        accountBalance = accountBalance + amount;
    }
}
```

- The `PayInFunds` method is given the amount of money to add to the balance
- It adds this to the `accountBalance` value in the account
- The method is `public`

Using the PayInFunds Method

```
Account Rob = new Account ();
Rob.PayInFunds(100);
Rob.PayInFunds(50);
```

- Users of the `Account` class can call the `PayInFunds` method to pay money into the account
- Each time the method is called the `accountBalance` value in the account is updated

A GetBalance Method

```
class Account
{
    private decimal accountBalance;

    public decimal GetBalance ()
    {
        return accountBalance ;
    }
}
```

- The `GetBalance` method returns the value of the account balance
- Note that this does not provide access to the field, instead it provides a copy of the value

Testing

- All our behaviours need to be tested
  - Particularly in terms of their error conditions
- Whenever we create a behaviour we should also create tests for that behaviour
- These tests should run completely automatically
  - The program should test itself

Testing the Account object

```
Account Rob = new Account ();
Rob.PayInFunds(100);
Rob.PayInFunds(50);
if (Rob.GetBalance() != 150 )
{
    Console.WriteLine ( "Test Failed" );
}
```

- As soon as we have some behaviours in our object we can write some tests for this object
- This one tests the `PayInFunds` and `GetBalance` methods

How Many Tests?

- The test seems quite sensible, but it is not sufficient to prove that the Account class works correctly
  - If `GetBalance` always returned 150 this test would pass
  - If every account was created with 150 pounds in it, and `PayInFunds` did nothing this test would pass
  - The test doesn't test a very good range of input values for `PayInFunds` - paying in a value of less than 0 is possible
- Whenever we create a behaviour in a class we should consider how it will be tested
- In many projects the tests are written first

Another Example of Test

- We must provide a method that withdraws funds from the account
- You tell it how much you want, and it either withdraws the money or tells you it can't
- The method will be called by other parts of the bank system when the customer uses their account:
  - When they use a cash machine to withdraw money
  - When they withdraw money at a bank branch

WithdrawFunds Method

- The method is called to withdraw money from the account
- It is given the amount of money to be withdrawn
- It returns true or false:
  - True means that the withdrawal succeeded and the cash can be released
  - False means there was not enough money in the account

A Potential WithdrawFunds method

```
public bool WithdrawFunds ( decimal amount )
{
  if ( accountBalance < amount )
  {
    return false ;
  }
  accountBalance = accountBalance - amount ;
  return true;
}
```

- A programmer has written this `WithDrawFunds`
- It seems sensible, but is it good enough?

Testing WithdrawFunds

```
Account Rob = new Account ();
Rob.PayInFunds(100);
if ( Rob.WithdrawFunds(60) == false )
    Console.WriteLine ( "Withdraw Test Failed" );

if ( Rob.GetBalance() != 40 )
    Console.WriteLine ( "Balance Test Failed" );
```

- This code creates an `Account`, pays in some money and then withdraws some
- If the withdraw fails, or the incorrect amount is left in the account the program prints error messages

Other Tests

- This is not a very good set of tests
- There are lots of other ones that will be required
  - Withdrawing an amount of 0
  - Withdrawing a negative amount
  - Withdrawing exactly the amount of money in the account
  - Making sure that the amount in the account only goes down when the withdraw succeeded
- This means that we will write more code in the tests than we wrote to implement the behaviour
  - This is perfectly normal

Preparing for the Worst

- It should not be possible for anyone (including other programmers) to be able to upset our bank account:
  - Never have a balance lower than the overdraft
- This is called *defensive programming* or *secure programming*
- Once we have our member information we now need to make sure that we look after it

Test Driven Development

- In *Test Driven Development* the tests are written before the methods themselves
- The methods are initially empty
- Then, during the development the methods are filled in so that the tests are passed
- The tests are run regularly during the development, and particularly after a bug has been fixed (a bug fix usually adds two bugs)
- You should remember to charge the customer for this work too!

Designing with Objects

- A class can contain data fields which it manages
- The data fields can be made private to protect them from code outside the class
- The programmer then creates methods that provide read (access) and write (mutate) behaviours as required
- Every behaviour must have tests associated with it to prove that the behaviour works correctly