POWERING
POTENTIAL

CODING STANDARDS AND PRACTICES

THE VIRTUES OF A PROGRAMMER

Laziness

- Hates answering the same questions over and over, so writes good documentation
- Hates reading documentation, so writes code clearly
- Writes tools and utilities to make the computer do all the work

Impatience

- Hates a computer that is lazy - an impatient programmer's code anticipates a need

Hubris

- Has pride in programs that no one will criticise

THE GOLDEN RULE

- **Follow the standards of your organisation**

- Ask to see the coding standards / guidelines

Then be sure your code always uses:

- Consistent naming
- Effective commenting
- Proper and effective code formatting

GOOD PRACTICE

1. Remember ‘Rubbish in – rubbish out’.
2. Choose the smallest data type for the job.
3. Always assign and operate on like data types.
4. Remember floating point issues.
5. Use constants, not literal numbers, where possible.
6. Create variables with the shortest scope and lifetime.
7. Make sure that objects are allocated and available for release as soon as possible.
8. Use variables for one purpose only.
9. Make sure that functions only perform one task.
10. Make sure that classes have a single responsibility, and identify what that is.
11. Automate your tests!

NAMING CONVENTIONS

Rules for the naming of identifiers:

- For data types - structures, classes, etc.
- For data items - constants, variables
- For code fragments - functions, methods, libraries

Naming conventions makes code easier to read:

- Easier to distinguish your items from those of a 3rd party
- Helps avoid clashes with reserved words

Data variables and constants should be clearly named:

- Functions should state their intent
- Avoid vague functions names like **`calculate`** preferring more specific **`calculate_invoice_total`**

COMMENTS

- Always comment the intent of your code
- Don't comment self evident coding structures
- Make sure you comment work-arounds and quick fixes
- Always update comments when you update code
- Remove comments from scripts before release

FORMATTING

Good formatting makes code clearer!

- Format your code to allow your code to naturally flow
- Languages like Python use indentation to specify blocks
- C Based languages use braces

```
foreach (string name in names)
{
    if (name.length < 5)
    {
        Console.WriteLine("Name isn't long enough");
    }
}
```

READABILITY AND STYLE

- More time is spent on maintenance than development
  - Document what you do
  - Code that is obvious today is not obvious tomorrow
- Avoid 'clever' one-liners
  - They are rarely faster, sometimes slower
  - Often difficult to debug
- KISS - Keep It Simple and Straightforward

ERROR HANDLING

- If anything can go wrong - it will
  - Specifically test error conditions
- If an opportunity exists to test for an error - take it
  - When calling a library routine
  - When getting any data from the outside
- Always report the error to an expected location
  - Write errors to an error stream, not the normal output stream
  - Users probably don't need the full story, but make sure it’s available to those that do
- It’s a common hacker's trick to cause a program to fail
  - Can result in the display of sensitive data, or the opening of a backdoor

PROGRAMMING FOR CHANGE

- Defensive programming
  - Where changes are less likely to cause problems
  - Includes good naming conventions
  - Clarity of style makes the code easy to read
- Making your program flexible
  - Avoid artificial fixed limits
  - Make no assumptions on variable type sizes, word length, etc.
  - Adhere to standards as much as possible
    - Using language extensions ties your program to one vendor
- All this makes changing a program less work
  - Great for the virtue of laziness!

HELP!

- Many languages have style checkers and rules
  - Python has PEP 008 rules
  - Perl has perlstyle documentation and PerlCritic
  - Java has Sun's code conventions
  - Google Style Guides are available for several languages
- Code analysis tools are often based on Lint
  - An old C-based tool, now much enhanced
  - Microsoft Visual Studio includes Code Analysis tool FxCop
- Some editors and IDEs assist with style as you type

Code Smells

COMMON CODE SMELLS

Long methods
Make code hard to maintain and debug – consider breaking up into smaller methods

Refuse bequest
When a class inherits from a base class and doesn't use any of the inherited methods

Data clumps
When multiple method calls take the same parameters

Duplicate code
You fix a bug, only for the same bug to then resurface somewhere else in the code

Other common code smells include middle man and primitive obsession.

SOLID PRINCIPLES

S - Single Responsibility Principle

O - Open-Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D - Dependency Inversion Principle

Please see the annex of this chapter for a revision of Solid Principles

LAB

Lab - **Coding Standards and Guidelines**

Afterwards, we'll discuss what you tried…

LAB

"Coding Style Guides" lab

Afterwards, we'll discuss your thoughts…

See also

Uncle Bob Martin's site http://cleancoder.com

CONTENTS

- **Objectives**
  - To understand the main principles of **SOLID**

Single-Responsibility Principle
Open/Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Follow these principals for good design and code

Single Responsibility Principle

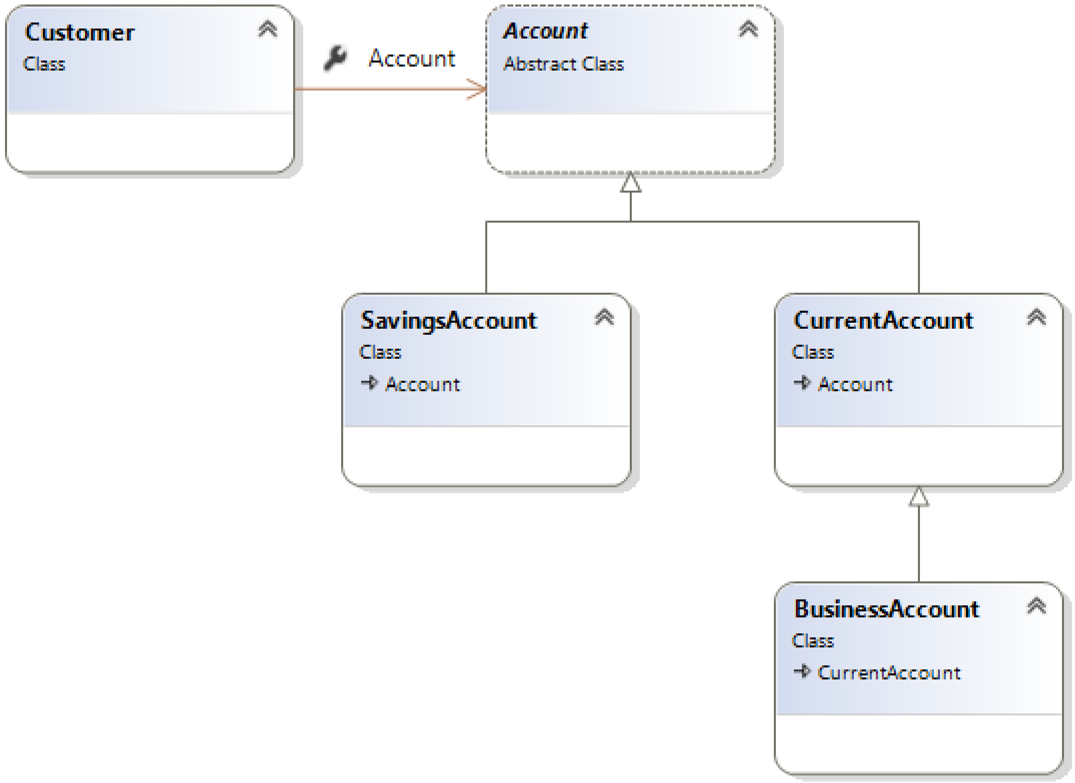
- **Each class should have only one responsibility**
- **Each class should have only one reason to change its behavior**
- **No *'Kitchen Sink'* classes.**
    - A class called QA which has all the methods needed to run QA!

- **This principal promotes decoupling.**
    - If you want to change an aspect, you'll have a lot less code changes
    - Code is reusable; by combining responsibilities other classes won't be able to use a single responsibility from the class.

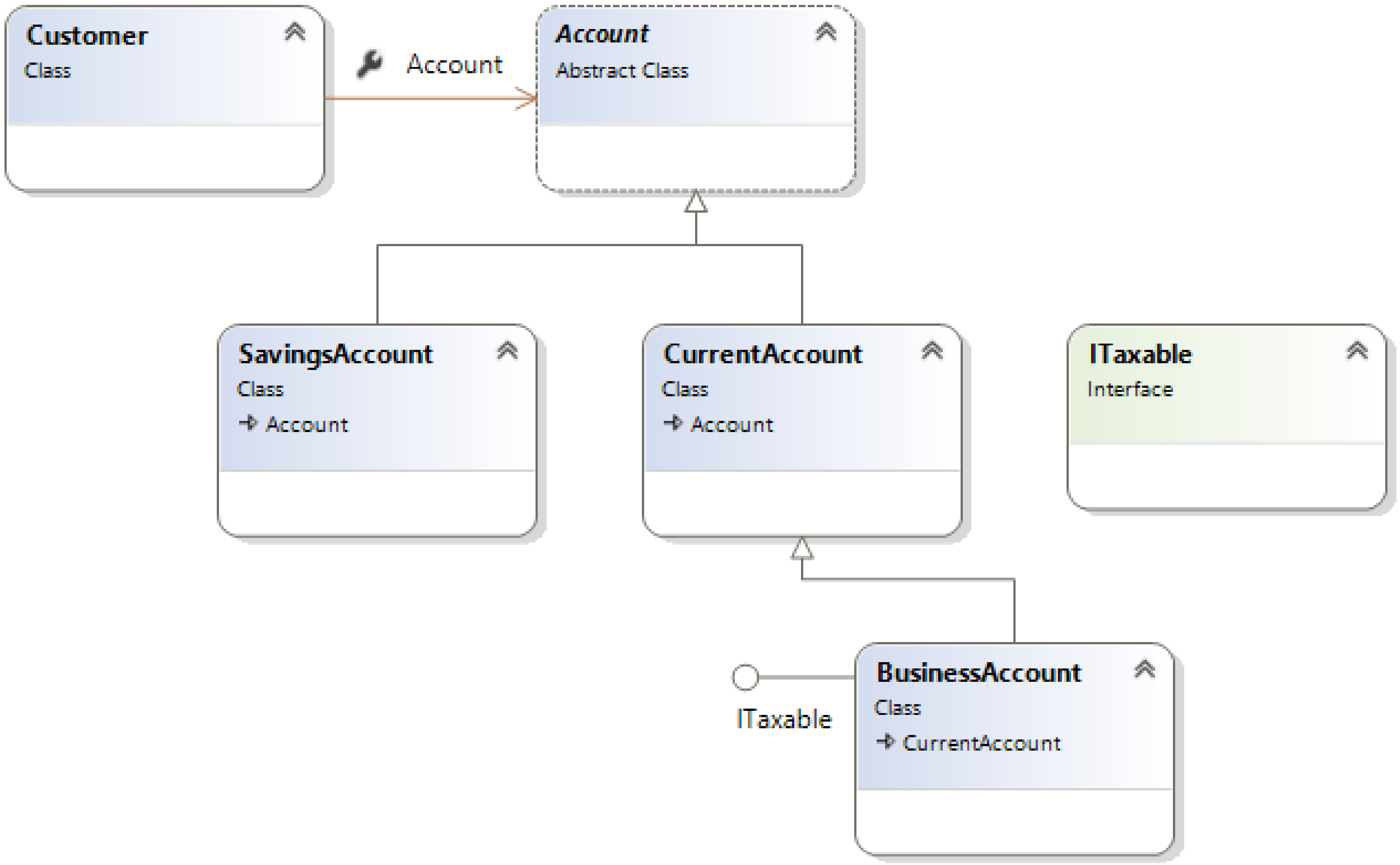
- **If you've a large class the chances are it's multiple responsibilities**
    - Just break it up into smaller classes which in turn makes the class easier to test.

Open Closed Principle

- **Open for extension, closed for modification.**
- **Software entities should be extendable, but not modifiable.**
- **Changes in requirements are performed using extension rather than modifying existing classes**
- **Apply abstraction and polymorphism.**

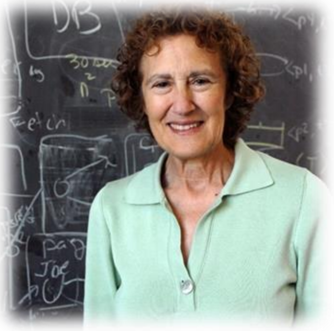


Open Close principal example



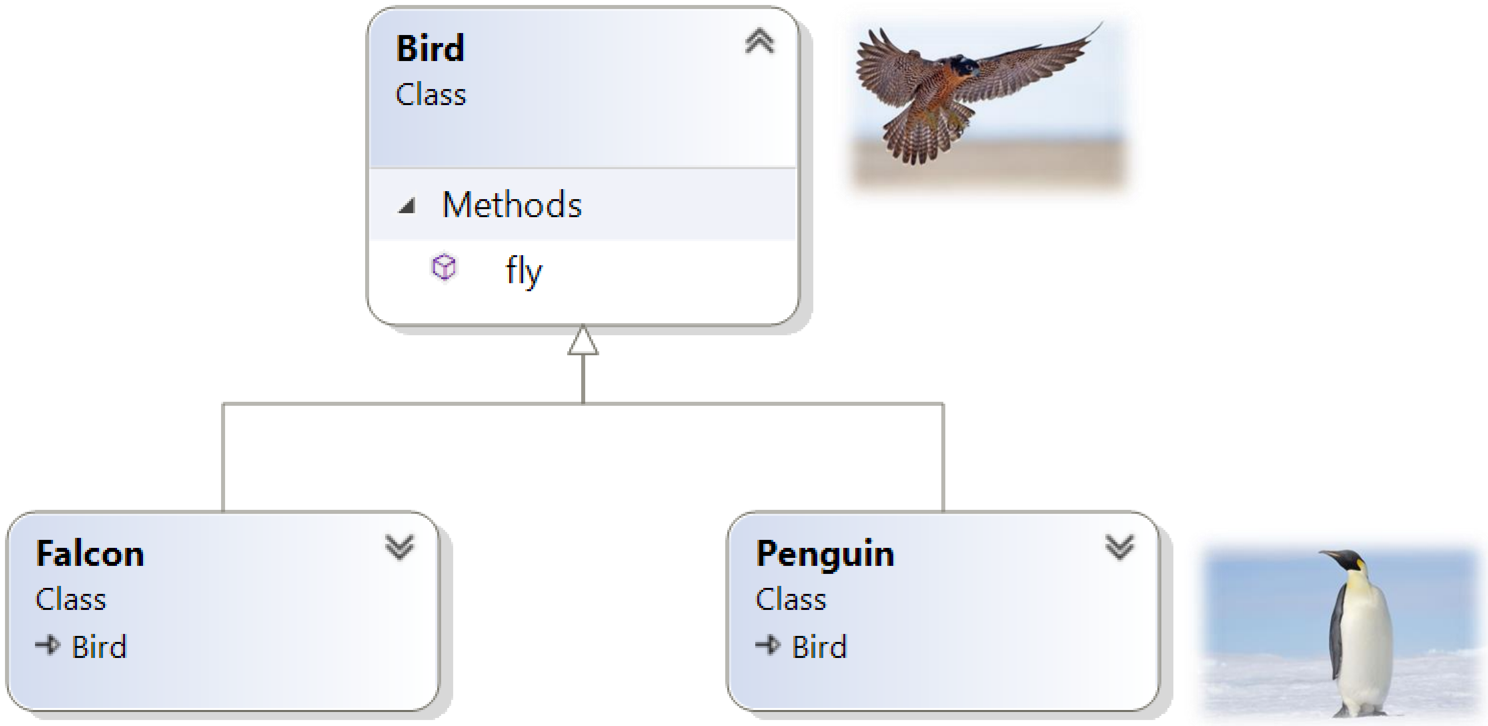
LISKOV SUBSTITUTION

- **Sub-classes behaviour should be the same as the super-classes**
  - Should fulfil all behaviours of the base class correctly.
- **The behaviour of a derived class should have a stronger post-condition and a weaker precondition than the base class.**
  - (you'll see this in the next few slides)
- **This principal prevents classes having undesirable behaviours.**



Barbara Liskov

LET’S BREAK LISKOV’S PRINCIPLE!



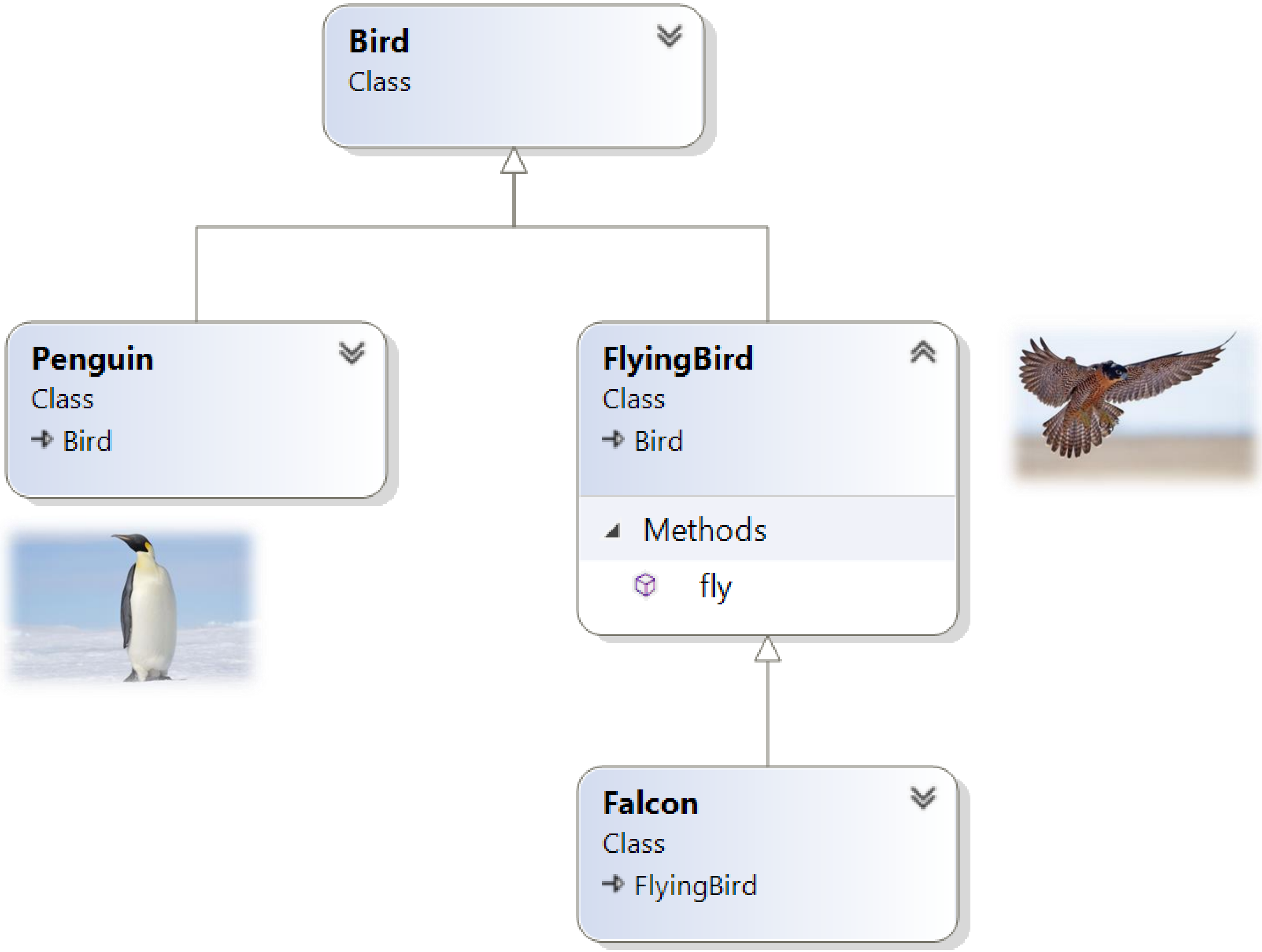
Bird
Class
Methods
fly

Falcon
Class
Bird

Penguin
Class
Bird

```
Bird[] birds = { new Falcon(), new Penguin(), new Falcon() };

for (int i = 0; i < birds.Length; i++)
{
    birds[i].fly();
}
```

LET'S APPLY LISKOV'S PRINCIPLE



Bird
Class

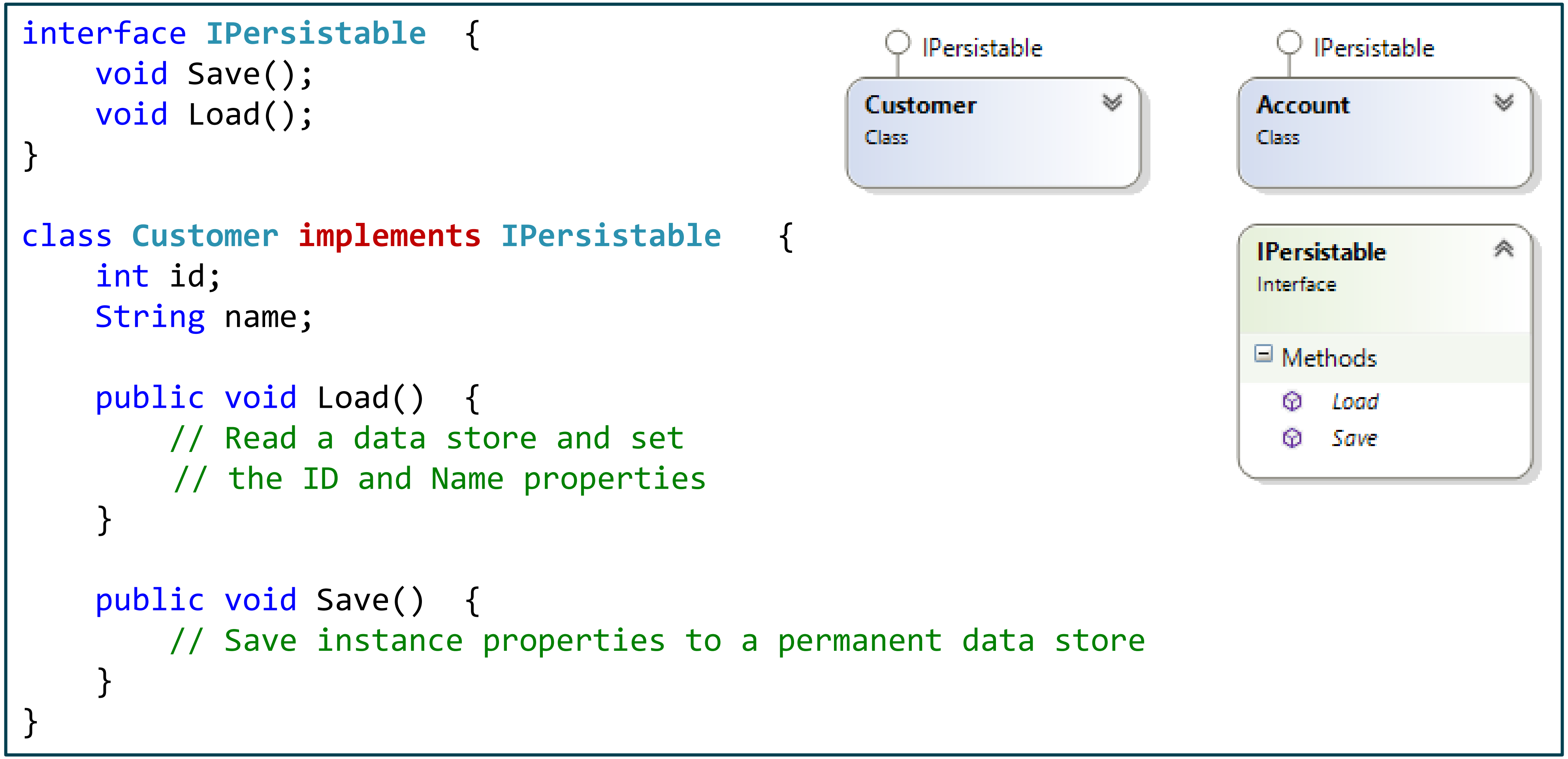
Penguin
Class
→ Bird

FlyingBird
Class
→ Bird

Methods
fly

Falcon
Class
→ FlyingBird

Another example of the Liskov Principle

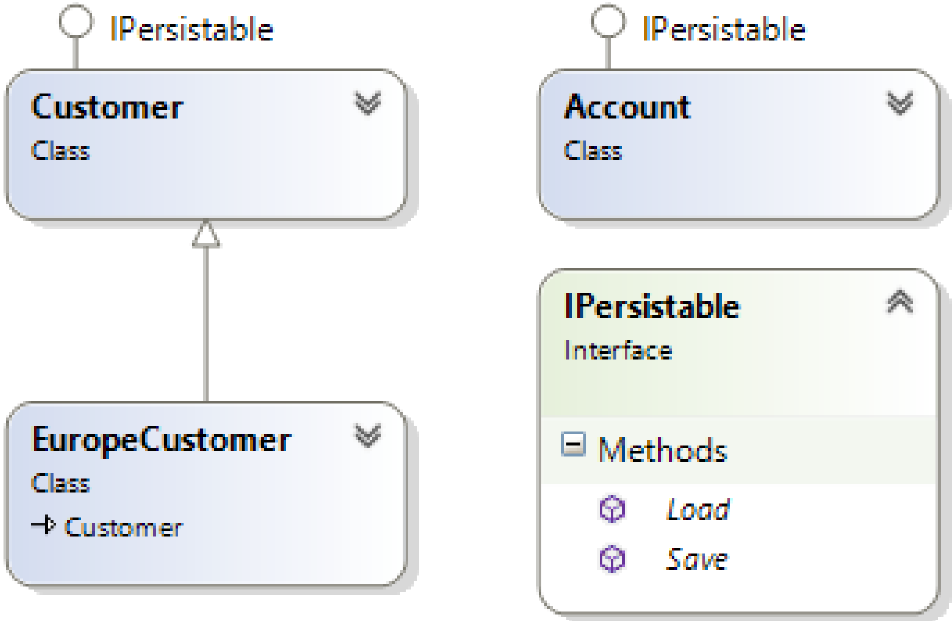


```
interface IPersistable  {
    void Save();
    void Load();
}

class Customer implements IPersistable   {
    int id;
    String name;

    public void Load()  {
        // Read a data store and set
        // the ID and Name properties
    }

    public void Save()  {
        // Save instance properties to a permanent data store
    }
}
```

Add a new class called EuropeCustomer

```
class EuropeCustomer extends Customer
{
    @override
    public void Save()
    {
        if (LocalTime.now().getHour() < 12)
        {
            // Save instance properties
        }
    }

}
```

What if the save() method is called in the afternoon?



Now let's try a new Customer class

- **What happens if one of the objects in the List is an *EuCustomer*?**

```
public class Manager
{
    public void SaveObjects(ArrayList<IPersistable> objects)
    {
        for(IPersistable item : objects)
        {
            item.Save();
        }
    }

    public void LoadObjects(ArrayList<IPersistable> objects)
    {
        for (IPersistable item : objects)
        {
            item.Load();
        }
    }
}
```

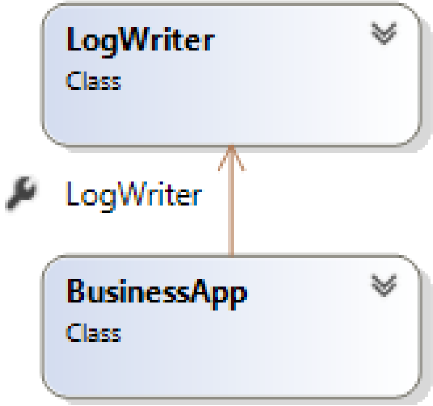
Dependency Inversion Principle (DIP)

- **High-Level modules should not depend on low-level modules – both should depend on abstractions**
- **Abstractions should not depend on details. Details should depend on abstractions.**
- **Instead of working with hard-wired / highly coupled classes you should always attempt to use interfaces**
- **Let's have a look at a simple example**

DI example – Bad design

- The following code logs message in a console
- Higher level class *(BusinessApp)* has a dependency on a concrete class *(LogWriter)*
- This violates the DI principle
  - What if you want to send email instead?

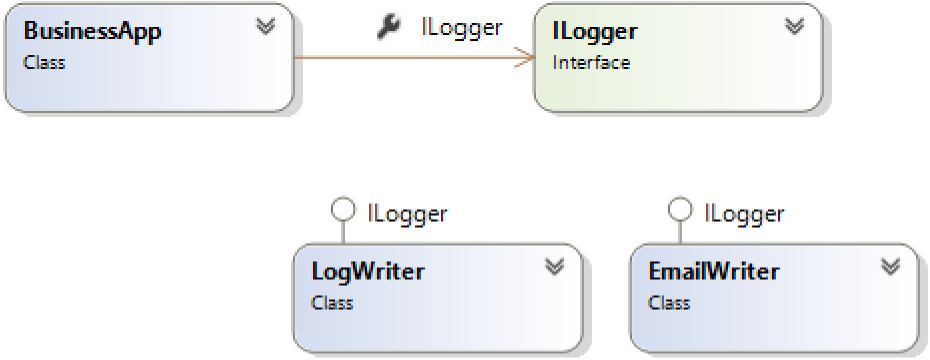
```
class LogWriter {
    public void Write(String message) {
        qa_WriteAllText("Log.txt", message);
    }
}

class BusinessApp {
    LogWriter logger = new LogWriter();

    public void Log(String message) {
        logger.Write(message);
    }
}
```



DI Example – Better design

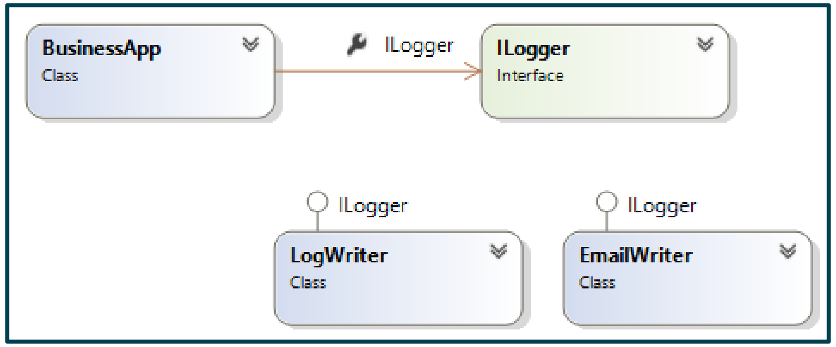
- **The BusinessApp now depends on abstractions (*ILogger* interface)**

- **BusinessApp has now got the freedom to Log messages using emails**



The C# Code for our DI first attempt

```
interface Ilogger {
    void Write(String message);
}

class LogWriter : Ilogger {
  public void Write(String message) {
     qa_WriteAllText("log.txt", message);
  }
}

class EmailWriter : Ilogger {
    public void Write(String message) {
        Outlook.Send("admin@qa.com", message);
    }
}
```

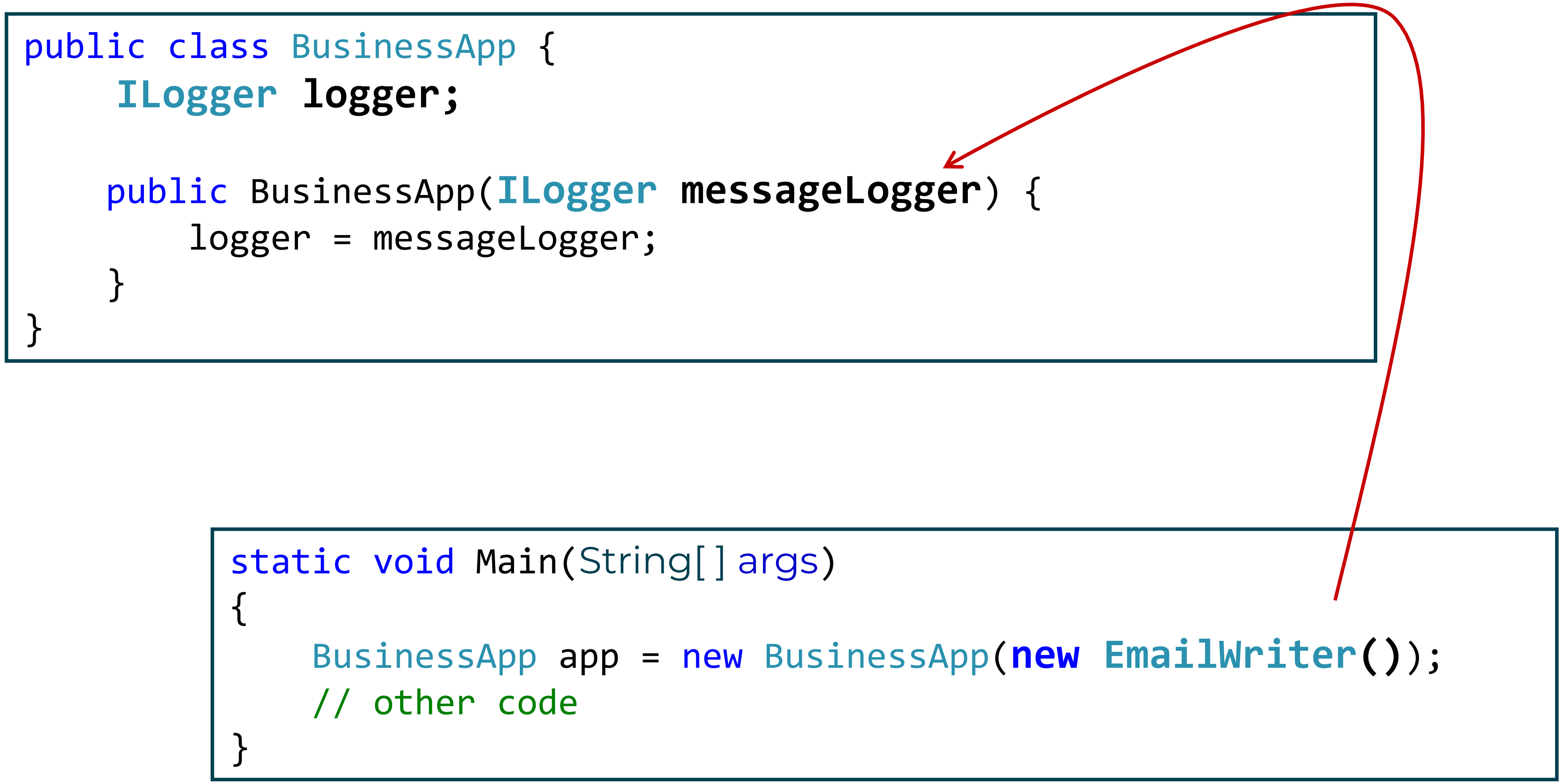


```
class BusinessApp {
    ILogger logger;
    public void Log(String message)  {
        if (logger == null)
            logger = new LogWriter();

        logger.Write(message);
    }
}
```

Could improve!

Dependency Injection

- Is for injecting the concrete implementation into a class that uses an abstraction (like an interface)
- You can inject dependencies in a Method/Constructor/Property



```
public class BusinessApp {
     ILogger logger;

    public BusinessApp(ILogger messageLogger) {
        logger = messageLogger;
    }
}
```

```
static void Main(String[] args)
{
    BusinessApp app = new BusinessApp(new EmailWriter());
    // other code
}
```

Dependency Injection an alternative

- Example shows how to inject a dependency in a Method

```
public class BusinessApp {

    public void Log(String message, ILogger messageLogger) {
        logger.Write(message);
    }
}
```

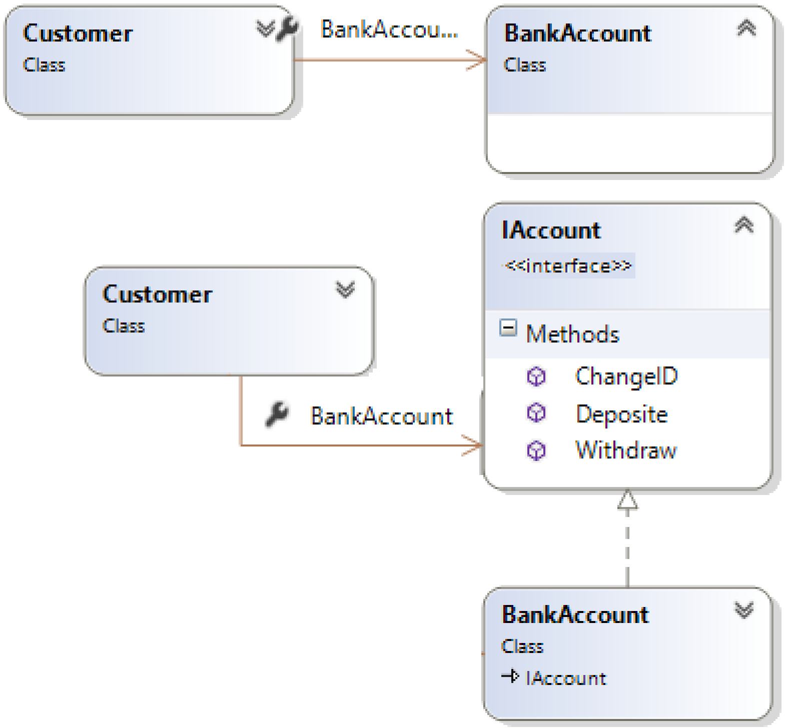
```
static void Main(String[ ] args)
{
    BusinessApp app = new BusinessApp();
    app.Log("Hello!", new EmailWriter());
}
```

INTERFACE SEGREGATION

- **Interfaces should be specific to a client. Many specific interfaces are better than one general purpose interface.**
- **By using specific interfaces, it will present a clear understanding on what behaviours the client should use.**
- **Additionally segregating interfaces increases usability of interfaces as interface implementers should only use behaviours that are relevant.**
- **Let's have a look at an example…**

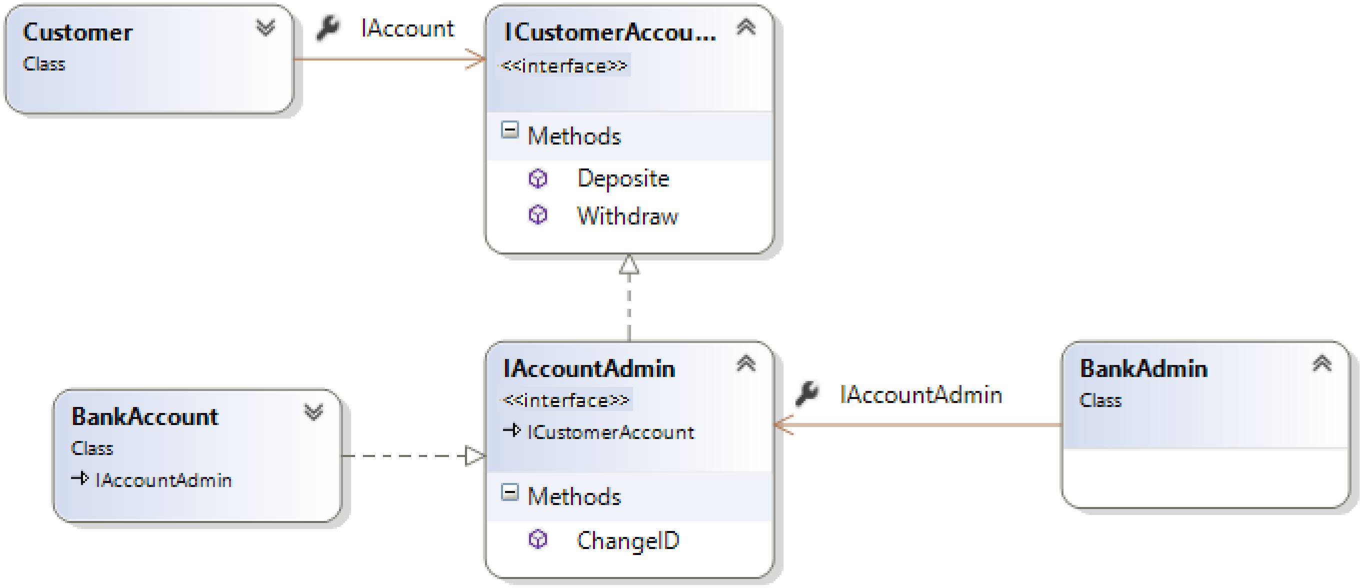
Customer association to a bank account

- First attempt: What could be the problem with this design?
- Second attempt: Is this solution better? Why?
- Yes! Customer can extend another class
- But what could be a problem with this solution?



Third attempt – Create many interfaces

- We should not allow customers to change their account ID!
- How would you solve this problem?
- The answer is to give customers their own account interface
- The bank should have the ability to change IDs and also have the same capabilities as a customer



Review

- **Symptoms of a degrading design**
  - Rigidity, fragility, immobility, viscosity
- **Degradation of dependency architecture**
- **SOLID**