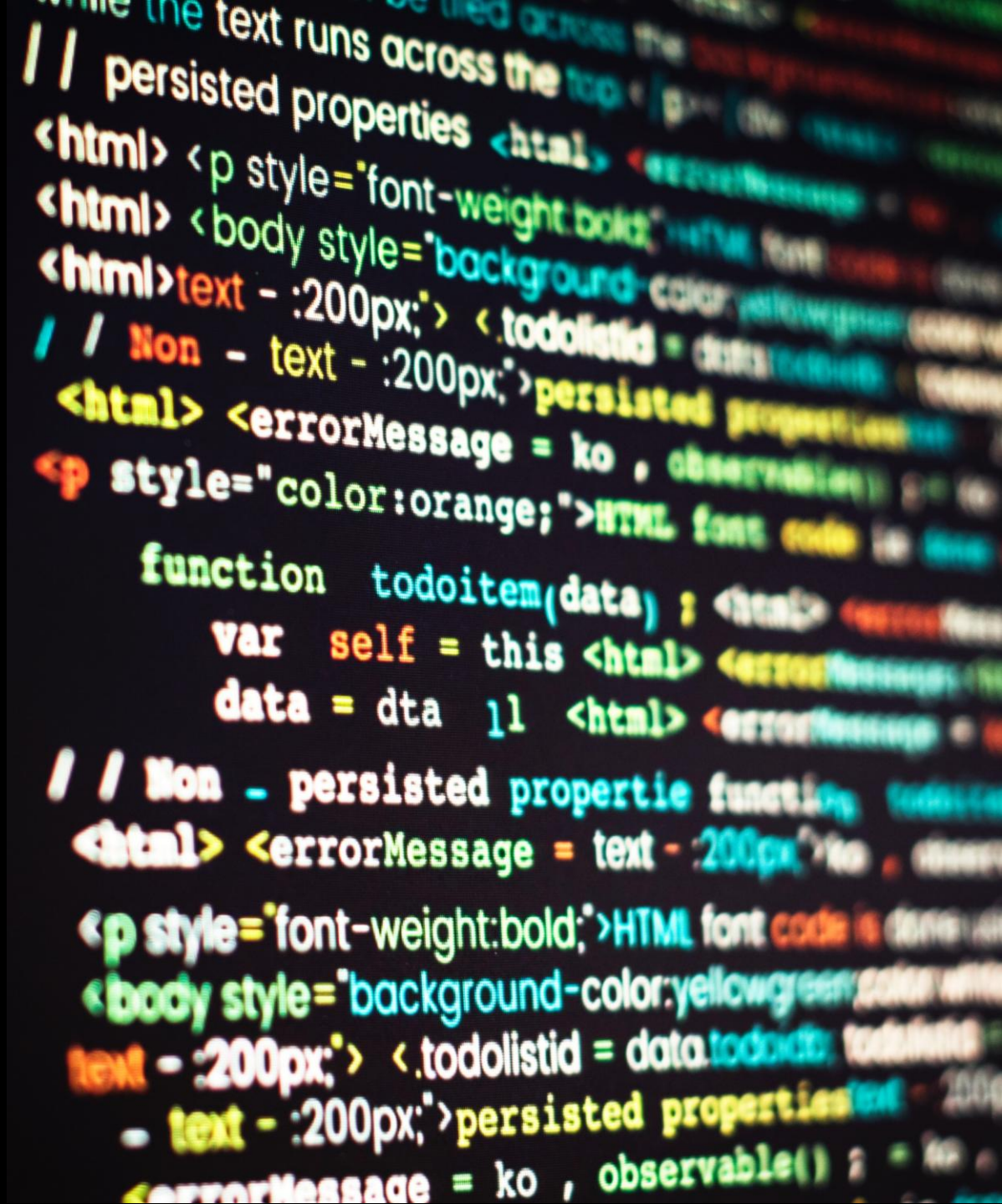
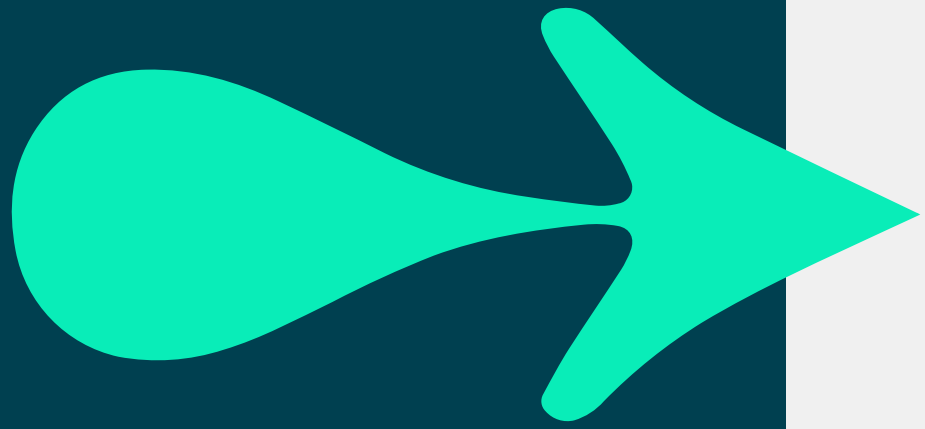
Code Smells

QA

Learn. To Change.



Module Objectives

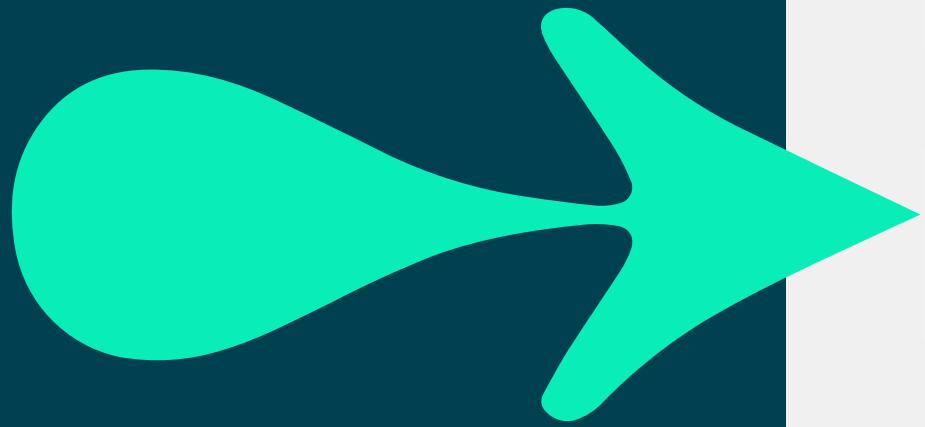


In this chapter we will explore

- What makes code smell!
- Code that may work and has no bugs but is:
  - **poorly written**
  - **Badly structured**
  - **Hard to read**
  - **Hard to maintain**
  - **Difficult to extend**.

- What steps can you take to overcome these problems?

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 most of the inherited methods

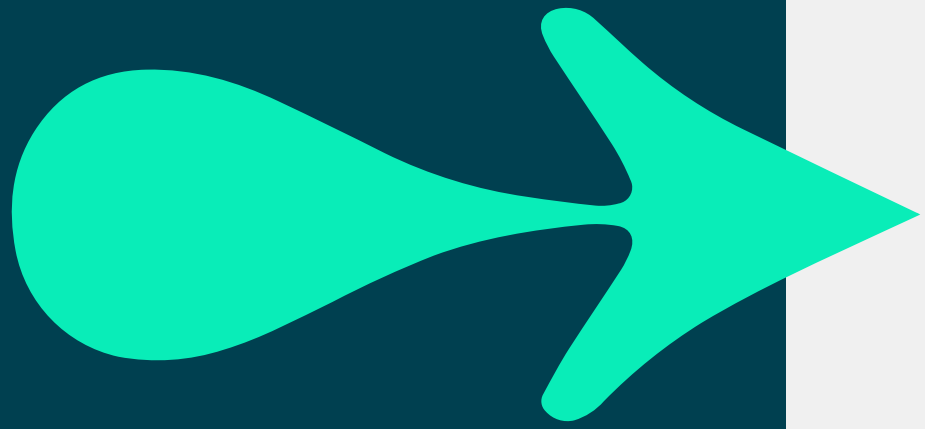
Data clumps

- When multiple methods calls take the same parameters

Duplicate code

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

Common Code Smells



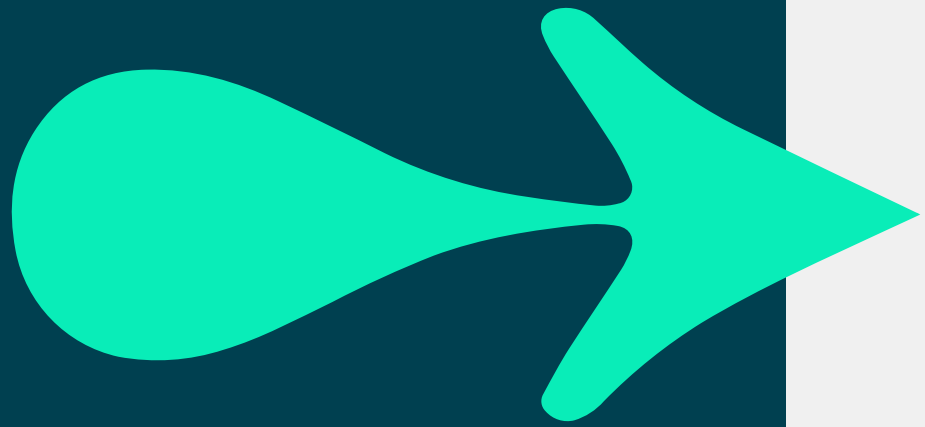
Middle-Man

- When a class does little more than delegate work to another class.

- Solution: Remove the unnecessary abstraction

```
class PaymentProxy {

    void makePayment(double amount) {

          QAPayment.processPayment(amount);

    }
}
```

Common Code Smells



Primitive Obsession

- Occurs when primitive types (e.g., strings, integers) are used instead of meaningful objects.

```
String customer="Bob,21,manager,Bristol";
```

- **Solution**: Replace primitives with well-defined classes

```
class Customer {
        String name, role, city;
        int age;

        // methods
}
```

SOLID Principles

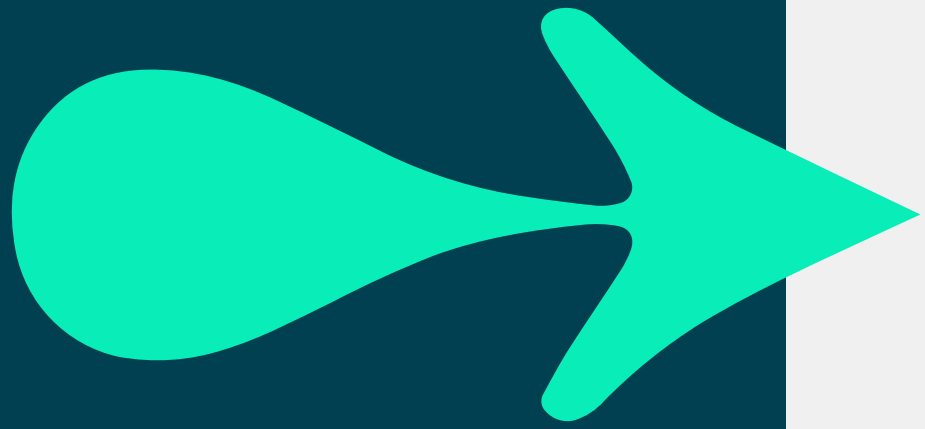
S - Single Responsibility Principle

O - Open-Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D - Dependency Inversion Principle



Follow these principles for good design and code

Single Responsibility Principle

- **Each class should have only one responsibility**
  - **Should have only one reason to change its behavior**
  - **No *'Kitchen Sink'* classes** like a class called QA which runs QA!

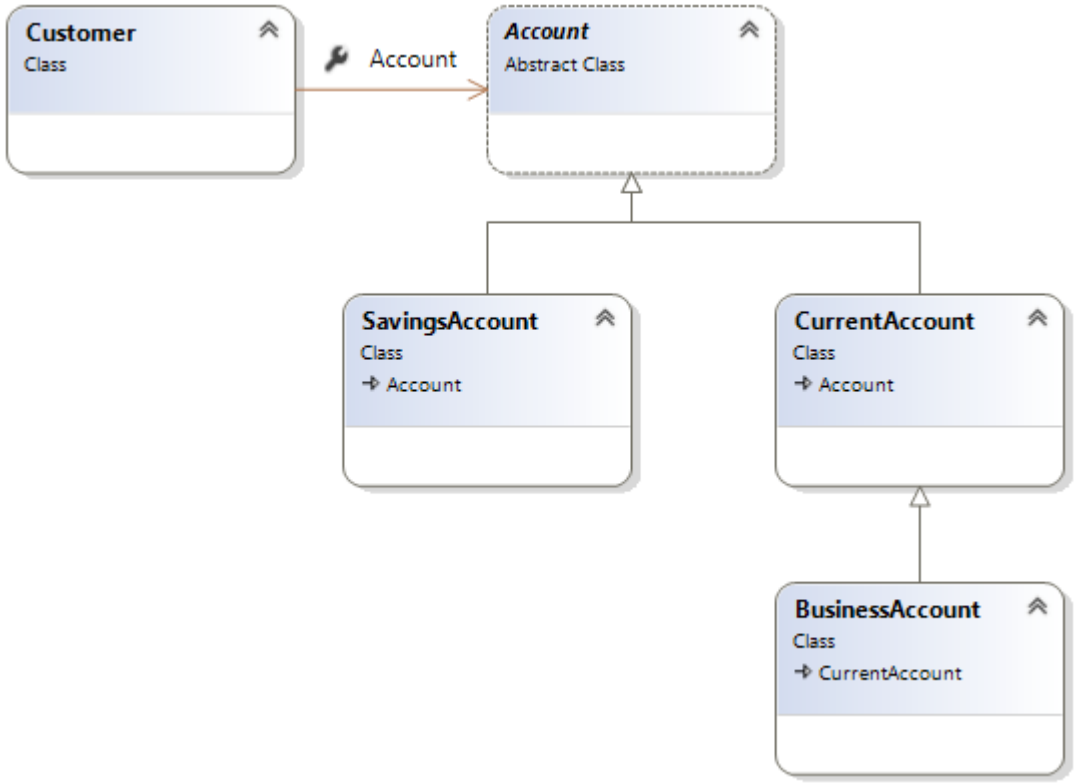
Queen
Worker
Drone
Guard
bees

- **This principal promotes decoupling.**
  - If you want to change an aspect, you'll have a lot less code changes
  - **Code is reusable**
    Other classes won’t be able to use a single responsibility from a class that has many responsibilities.

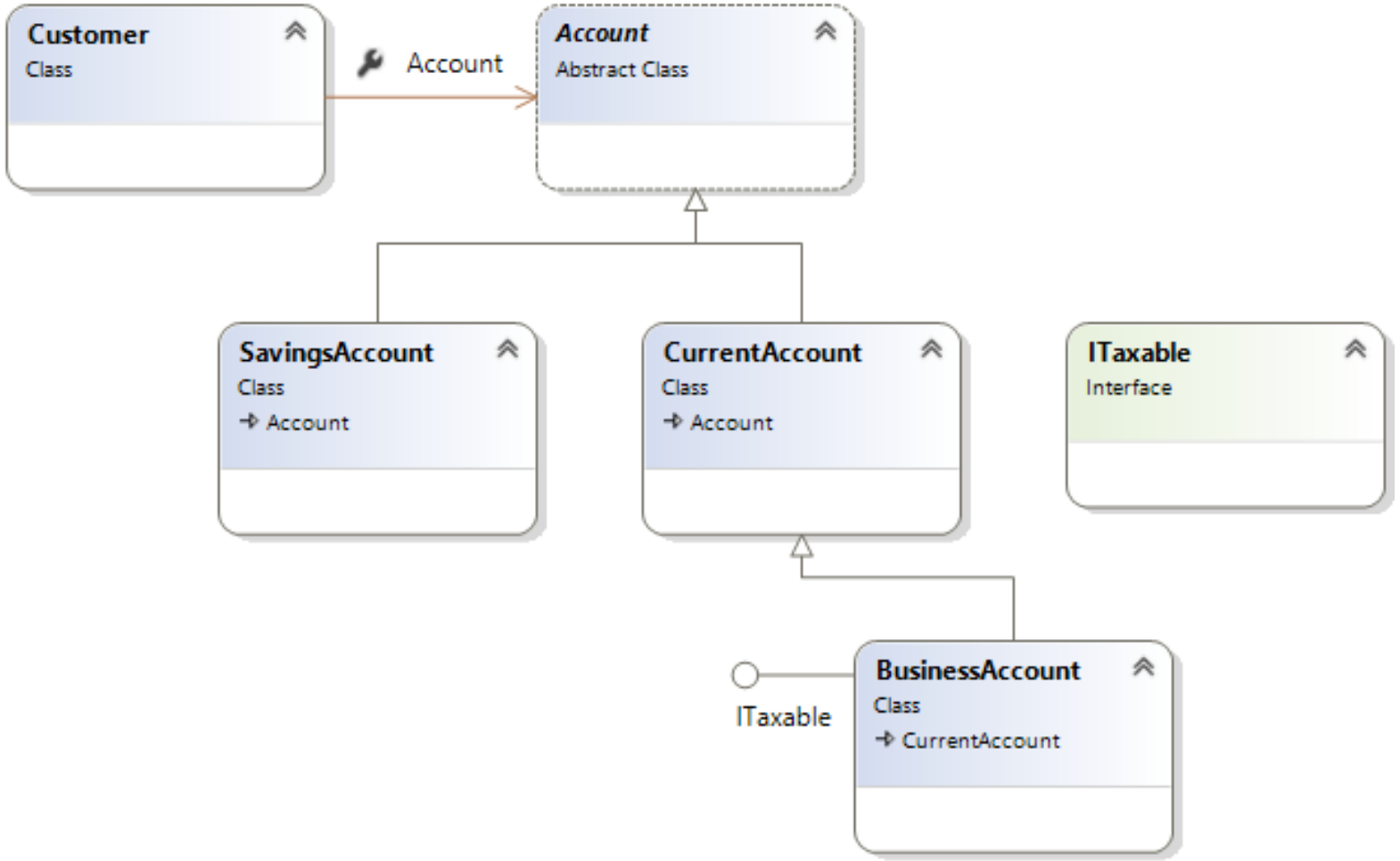
- **If you've a large class, the chances are it has multiple responsibilities**
  - Just break it up into smaller classes which in turn makes the class easier to test.

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 – use Interfaces



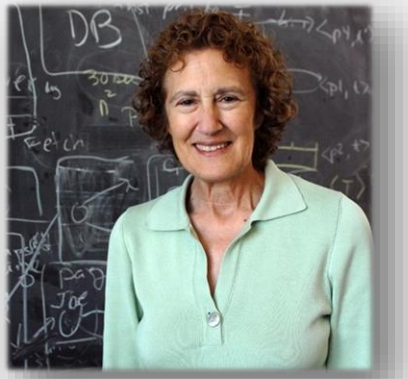
QA

Liskov Substitution

- **Sub-classes behaviour should be the same as the super-classes**

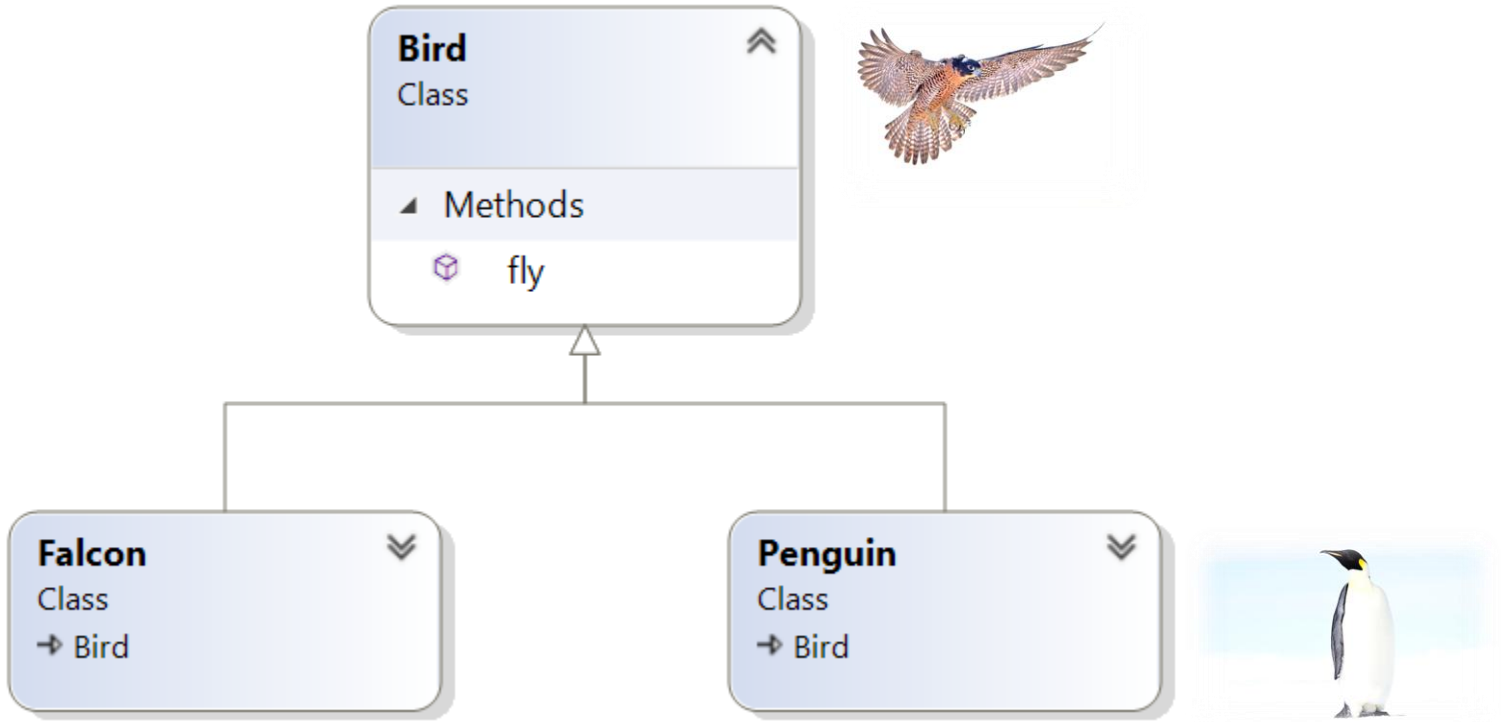
- **The behaviour of a derived class should have a stronger post-condition and a weaker precondition than the base class.**
  - Derived class works in any situation where the base can
  - Derived class can offer more but not less

This principal prevents classes having undesirable behaviours.



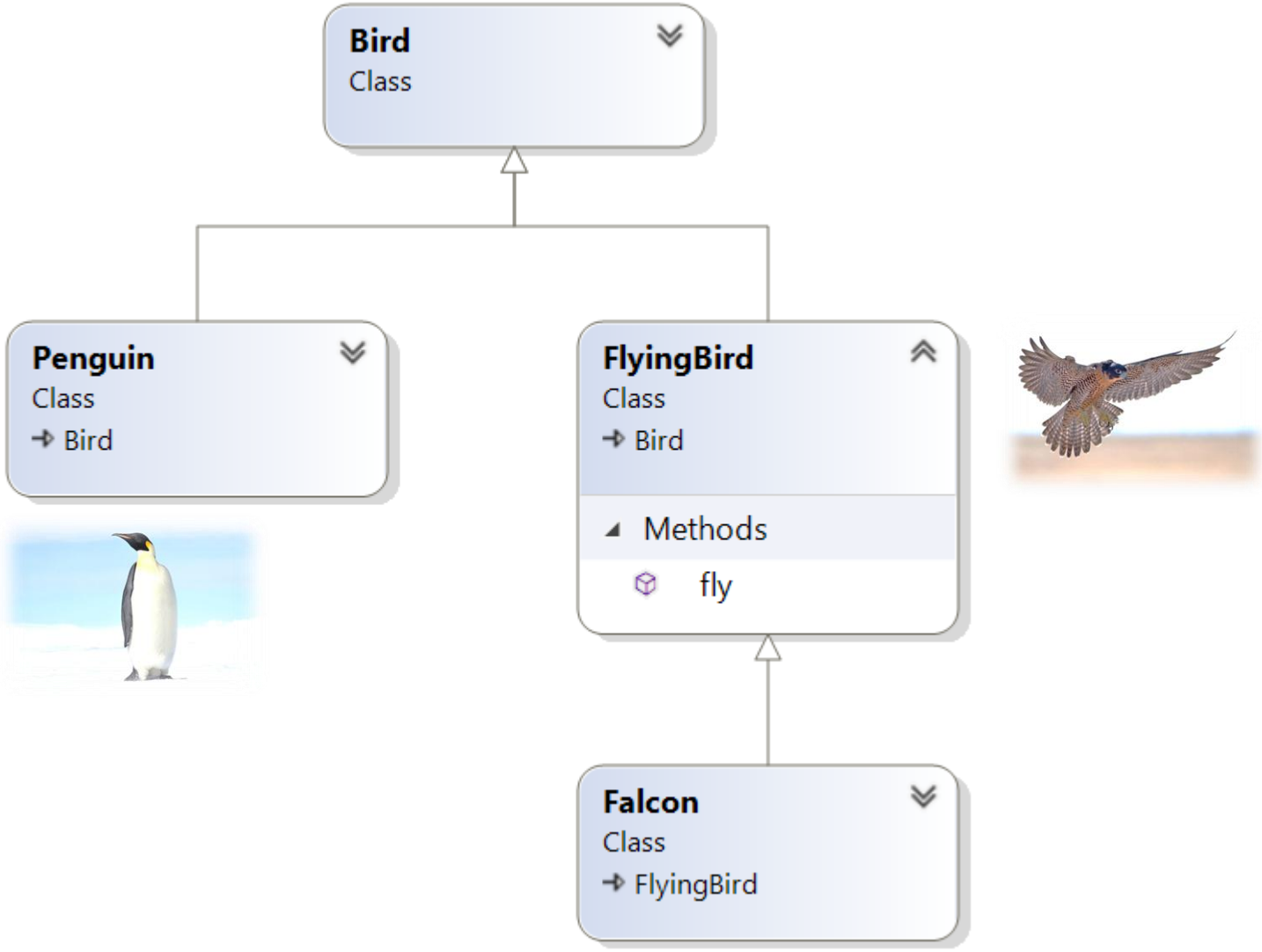
Barbara Liskov

Let’s break Liskov’s principle!



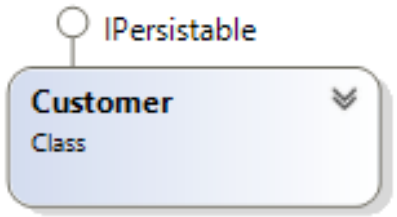

```
Bird[] birds = { new Falcon(), new Penguin(), new Falcon() };

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

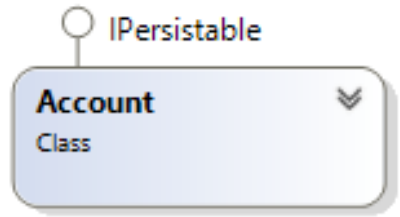
Let’s apply Liskov’s principle

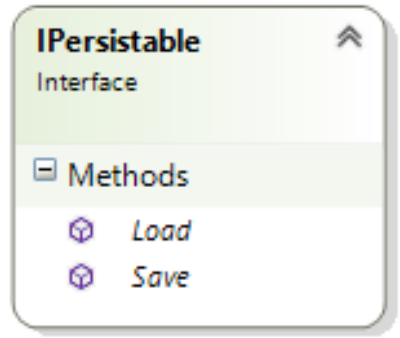


Another example of the Liskov Principle

```
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
    }
}
```



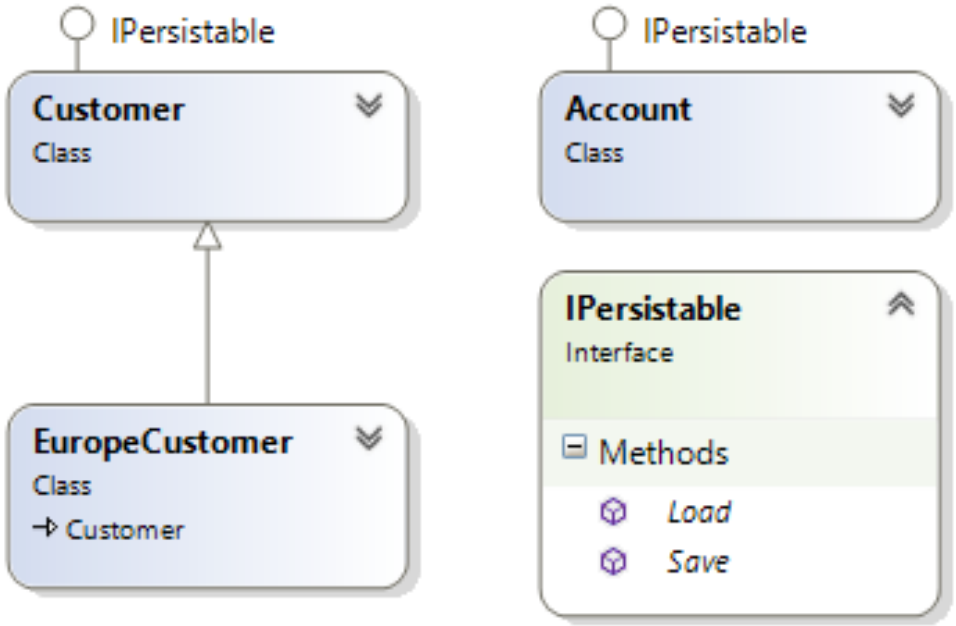




Add a new class called EuropeCustomer

```
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?

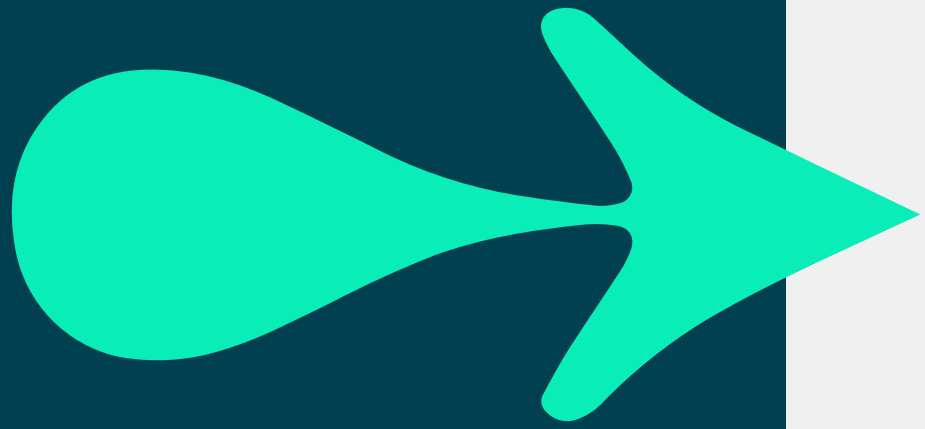


Now let's try a new Customer class

- **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 concrete implementation.

Instead of working with highly coupled classes you should attempt to use interfaces

Let's have a look at a simple example ➡️

Dependency Inversion example

```
class EmailService {
    void sendEmail() { ... }
}

class UserController {
    EmailService service = new EmailService();  // depends on concrete class
}
```

Not using DI

```
interface MessageService {
    void send();
}

class EmailService implements MessageService {
    public void send() { ... }
}

class UserController {
    MessageService service;     // depends on abstraction
}
```

Using DI

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());
}
```

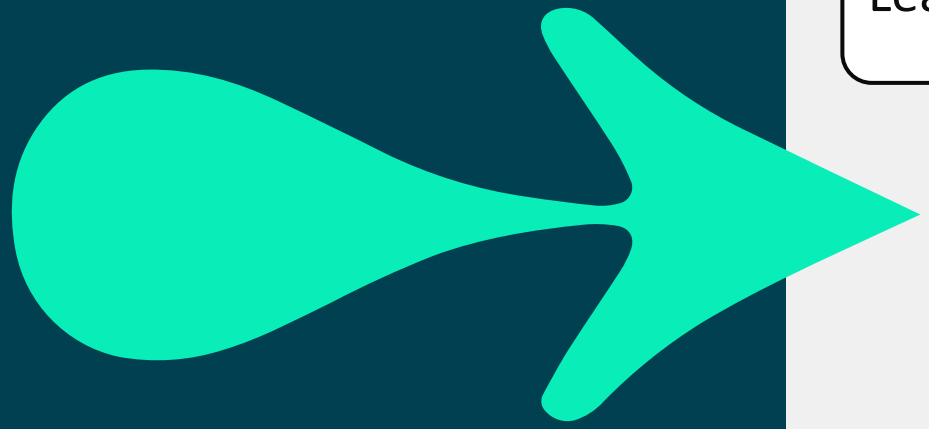
Interface Segregation

It's better to have **many small, specific interfaces** than one large, general-purpose one

Each interface should define only the behaviours relevant to its client
Makes code easier to understand, implement, and maintain

Don't have to Implement **unnecessary methods** you don't need

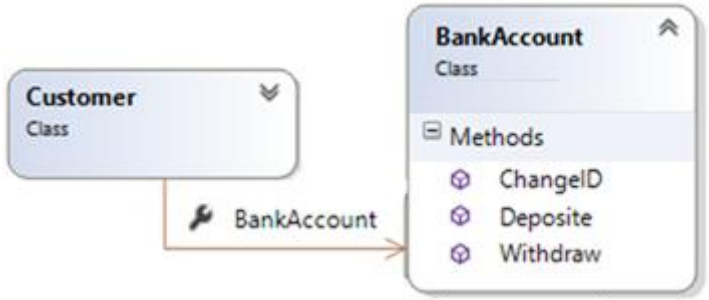
Leads to **cleaner, more modular design**



Let's have a look at an example…

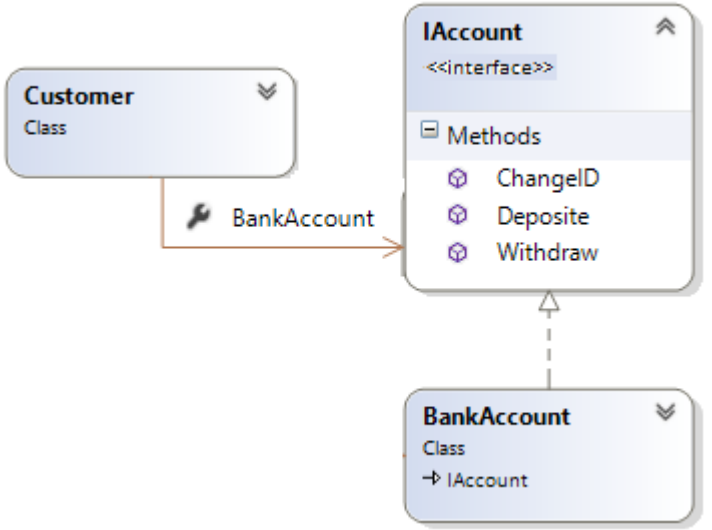
Customer association to a bank account

First attempt:
Any problem with this design?



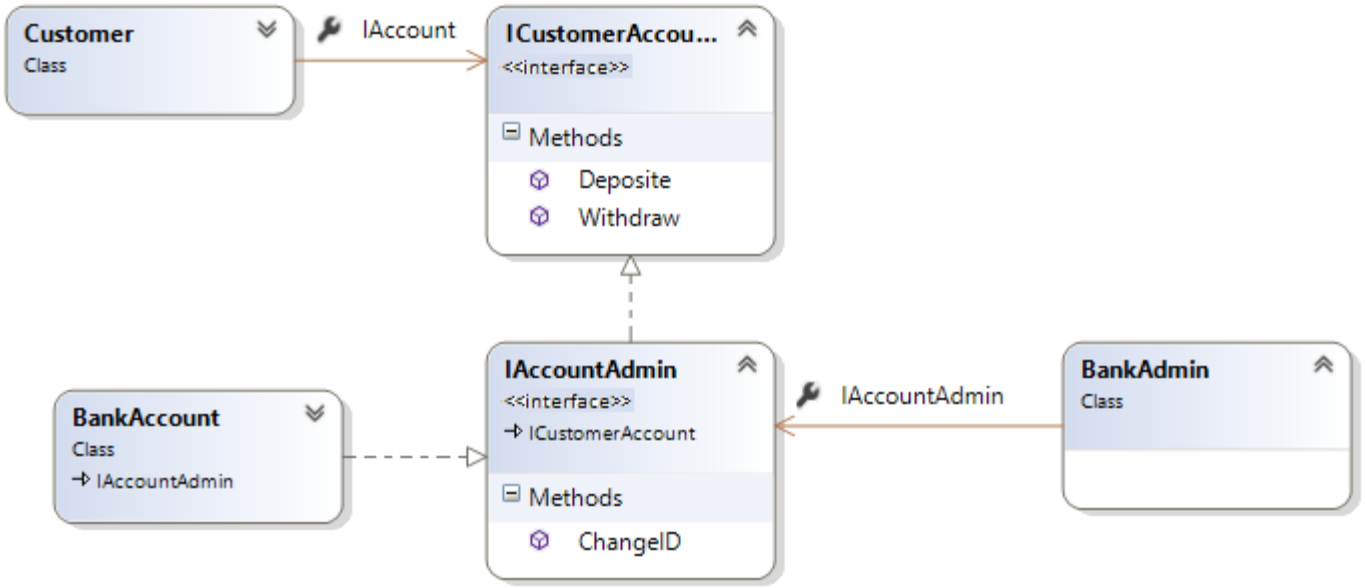
Second attempt:
Is this solution better?

Any problem with this solution?



Third attempt – Create many interfaces

- We should not allow customers to change their own IDs!
- Give customers their own account interface
- The bank admin should be able to modify customer IDs and have all the same capabilities as a regular customer.



Review

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

"Code Smells" Lab

Afterwards, we'll discuss what you tried...

QA

Learn. To Change.