SENG 350
- Software Architecture & Design

Shuja Mughal

Design Patterns

Fall 2024

University of Victoria

Web programming

- The non-design bits
- Developing web applications of all scales

  "In software engineering, a Web application or webapp is an application accessed with a Web browser over a network such as the Internet or an intranet. Web applications are popular due to the ubiquity of the browser as a client, sometimes called a thin client. The ability to update and maintain Web applications without distributing and installing software on potentially thousands of client computers is a key reason for their popularity. Web applications implement Webmail, online retail sales, online auctions, wikis, discussion boards, Weblogs, MMORPGs, and many other functions."

Webapps summary

- Accessed with a Web Browser (client)
- Over a network

- Code is mainly run on server
- Exception: Javascript (also: Java, Flash,..)

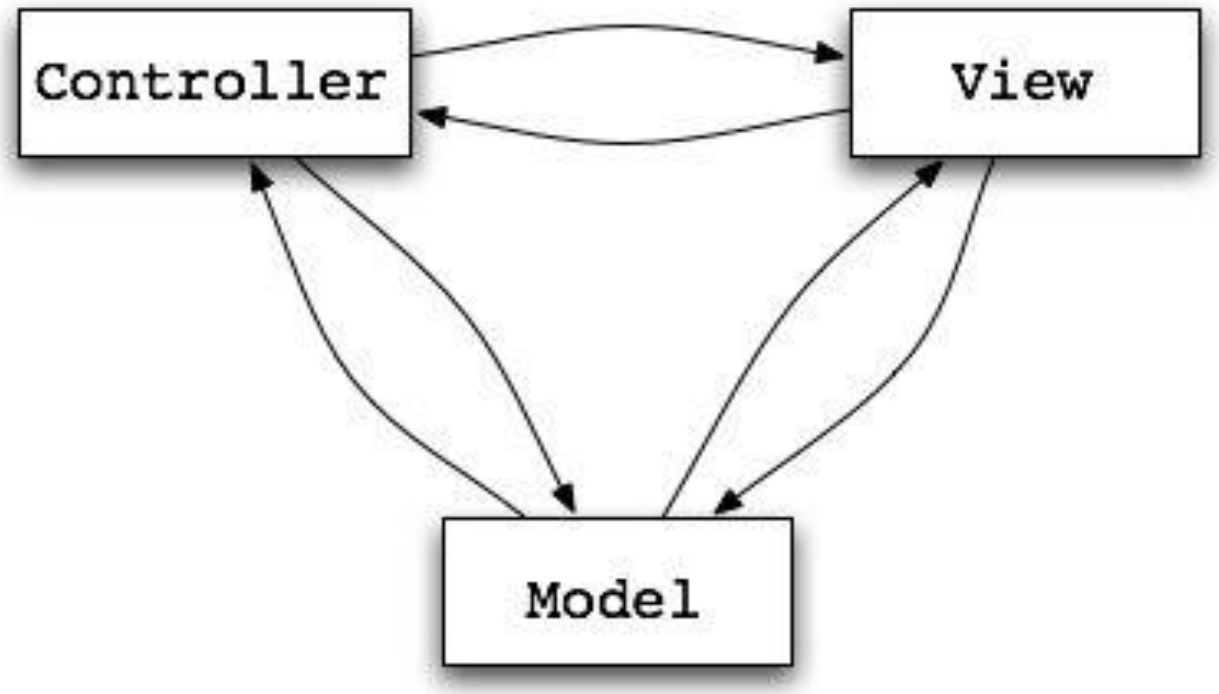
- Code is loaded from server
- Data is mainly stored on server

- Webapps can be updated easily…
  ..without updating the clients!

General 3 tiered structure

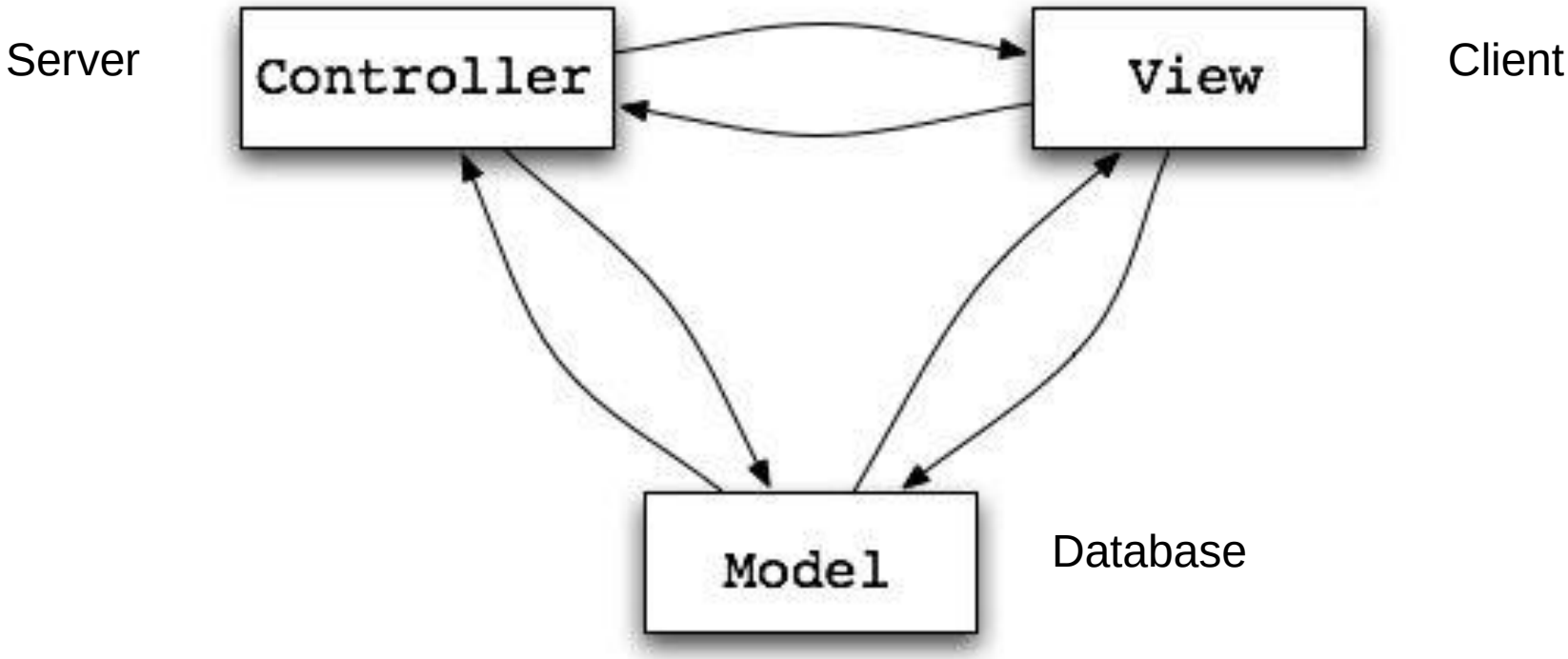
- First tier: client side code (web-browser), e.g. (X)HTML, Javascript, Java, Flash
- Second tier: server side code, e.g. C/C++, Perl, PHP, Java, Ruby, Python
- Third tier: server side database

Model View Controller



- Architectural Pattern from Smalltalk
- Decouples data and presentation
- Eases the development

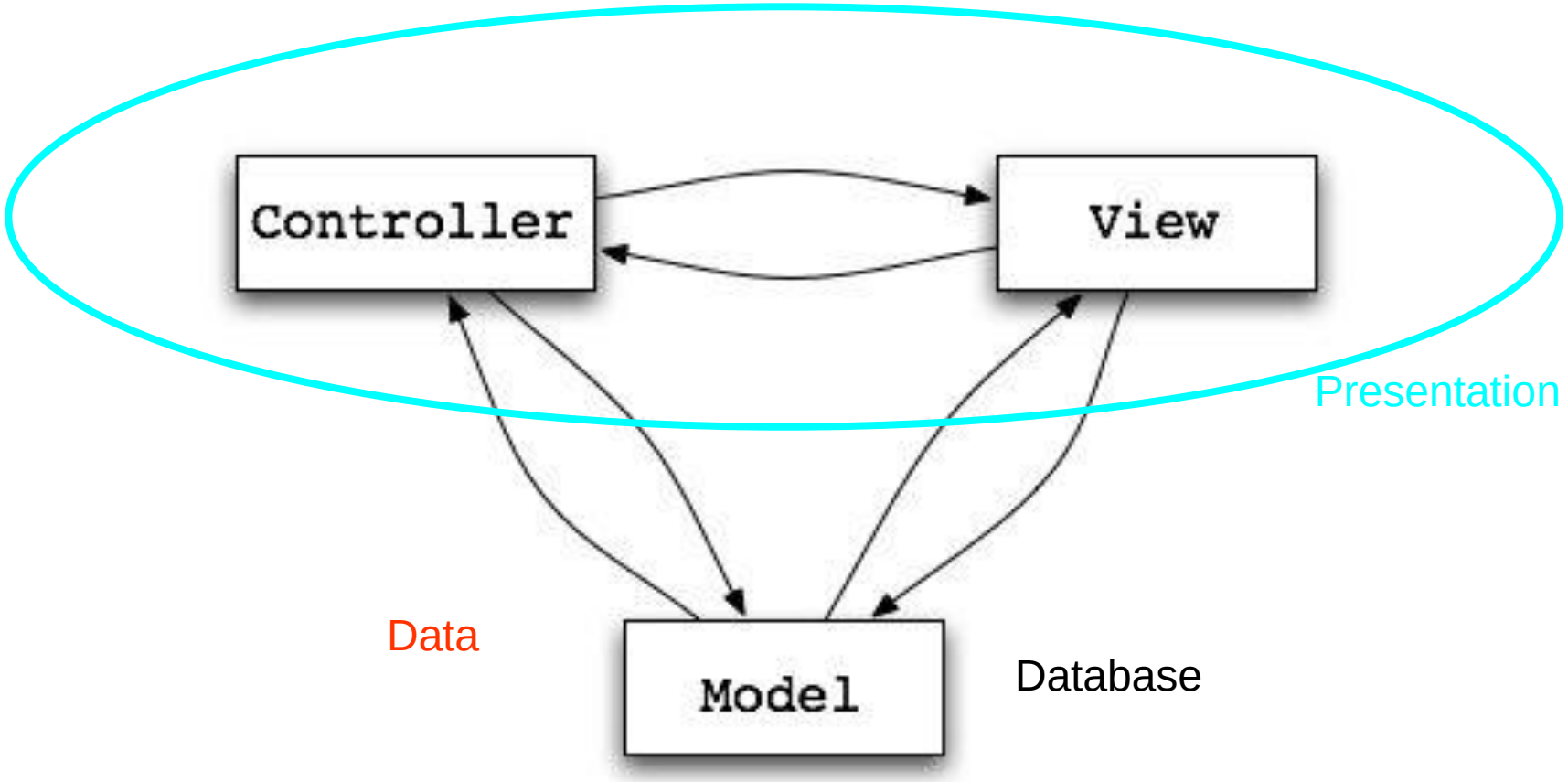
University of Victoria



First thought (ok, but not far enough):

- Tier 1: View (Client)
- Tier 2: Controller (Server)
- Tier 3: Model (Database)

University of Victoria



Presentation:

View is the user interface (e.g. button)

Controller is the code (e.g. callback for button)

Data:

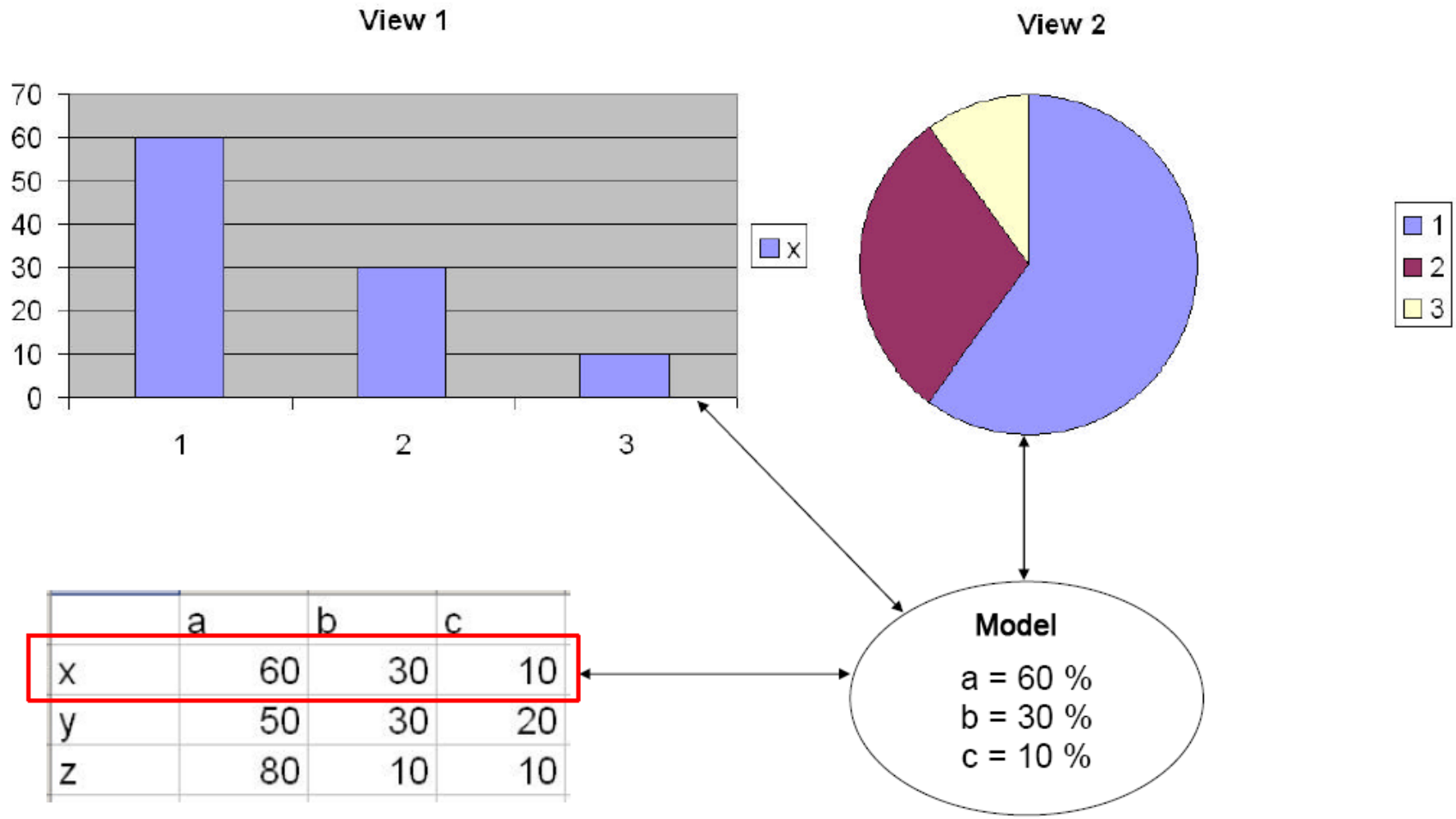
Model is the database

University of Victoria

Example Control Flow in MVC

- User interacts with the VIEW UI
- CONTROLLER handles the user input (often a callback function attached to UI elements)
- CONTROLLER updates the MODEL
- VIEW uses MODEL to generate new UI
- UI waits for user interaction

MVC – general example



Design Patterns

By Purpose				
		Creational	**Structural**	**Behavioral**
By Scope	**Class**	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Singleton Pattern

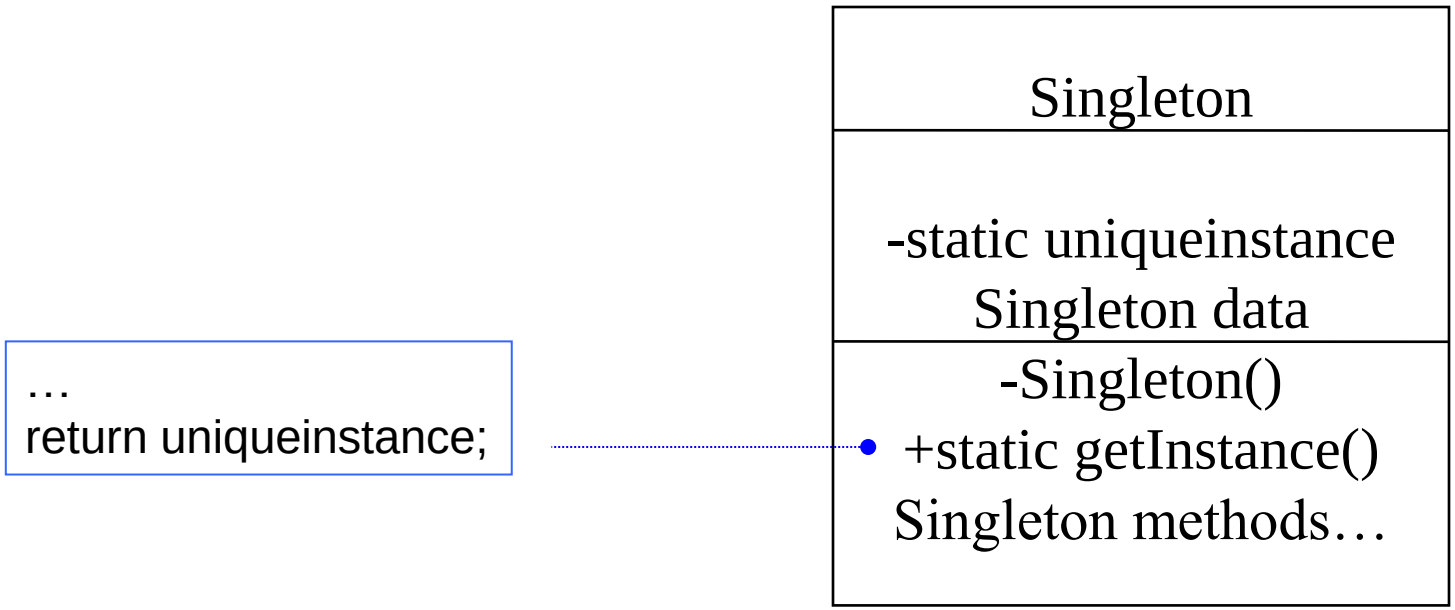
Singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it
- Motivation
  - Important for some classes to have exactly one instance.
  - Ensure only one instance is available and easily accessible
    - global variables give access, but don’t keep you from instantiating many objects
  - Give class responsibility for keeping track of its sole instance

Design Solution

Defines a getInstance() operation that lets clients access its unique instance

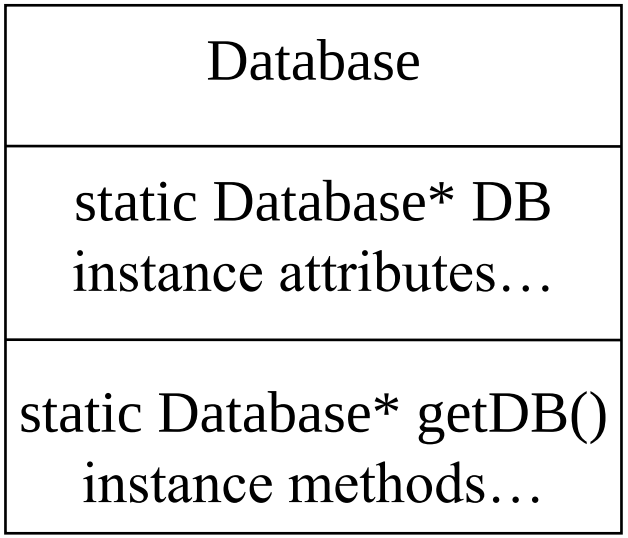
May be responsible for creating its own unique instance



University of Victoria

Singleton Example (Java)

Database




```
public class Database {
private static Database DB;
   ...
private Database() { ... }
public static Database getDB() {
   if (DB == null)
     DB = new Database();
   return DB;
}
   ...
}
```

In application code…

```
Database db = Database.getDB();
db.someMethod();
```

Singleton Example (C++)

```
class Database
{
private:
   static Database *DB;
   ...
   private Database() { ... }
public:
   static Database *getDB()
   { if (DB == NULL)
       DB = new Database());
     return DB;
   }
   ...
}
Database *Database::DB=NULL;
```

In application code…

```
Database *db =
       Database.getDB();
Db->someMethod();
```

University of Victoria

Implementation

- Declare all of class's constructors private
  - prevent other classes from directly creating an instance of this class
- Hide the operation that creates the instance behind a class operation (getInstance)
- Variation: Since creation policy is encapsulated in getInstance, it is possible to vary the creation policy

Singleton Consequences

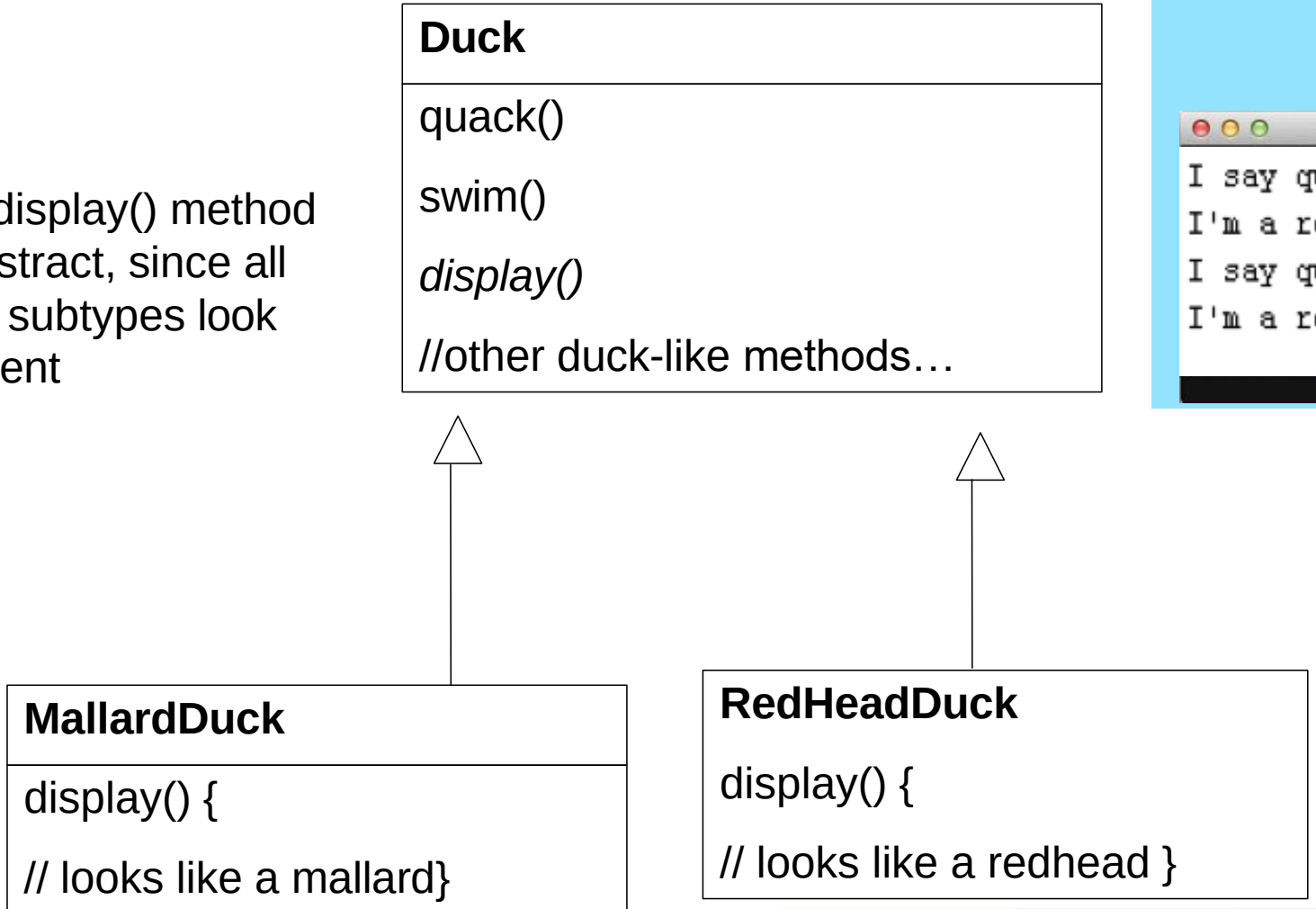
- Ensures only one (e.g., Database) instance exists in the system
- Can maintain a pointer (need to create object on first get call) or an actual object
- Can also use this pattern to control fixed multiple instances
- Much better than the alternative: global variables

By Purpose				
		Creational	**Structural**	**Behavioral**
By Scope	**Class**	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

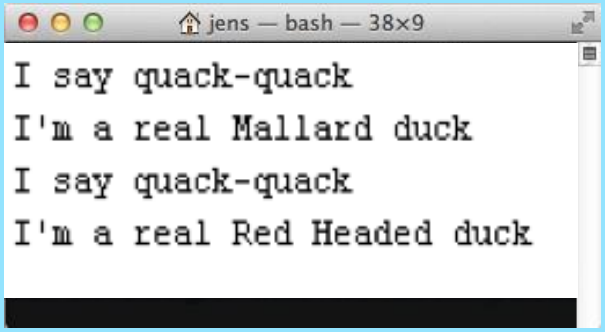
The Strategy Pattern

Introductory Example: (OO) Duck application

The display() method is abstract, since all duck subtypes look different



Test run:



University of Victoria

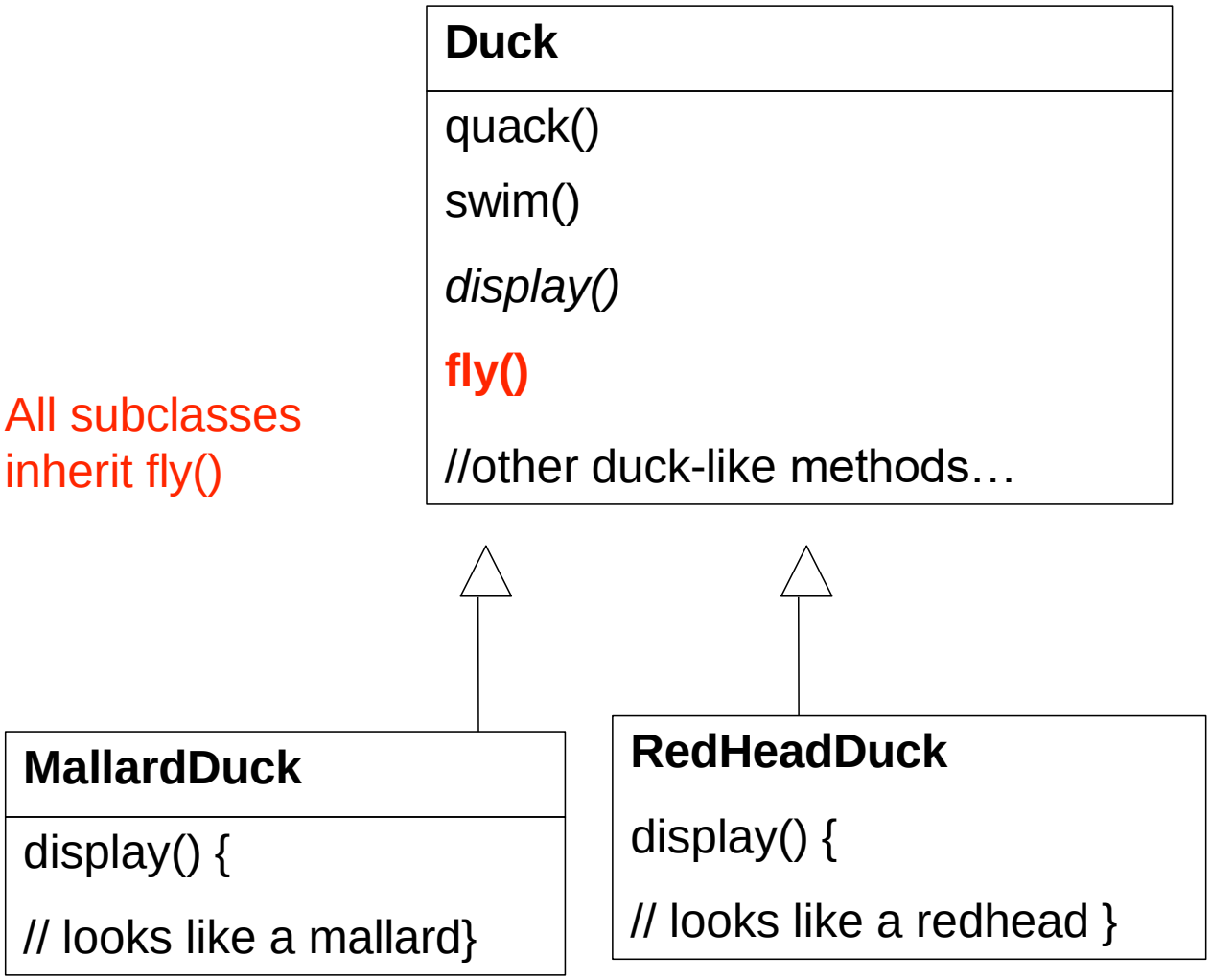
But Executives want to introduce fly capability quickly

How to do it in a short time, like 2 days?

University of Victoria

Solution #1: add a method fly() in Duck

All subclasses inherit fly()

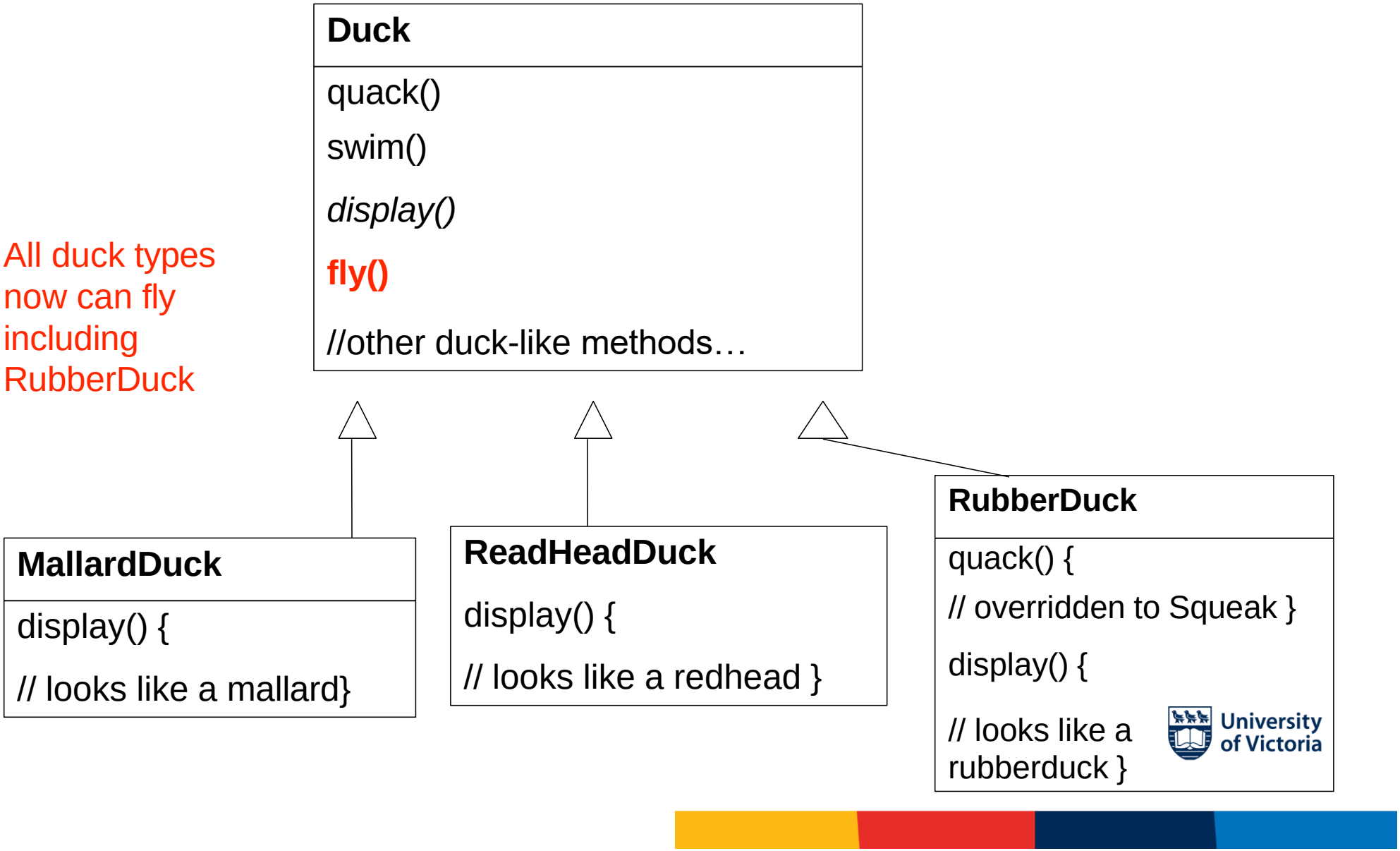


Later it turns out that there are exceptions.
Not all ducks can fly!!!!

University of Victoria

Something's wrong

All duck types now can fly including RubberDuck



How do we fix this?

How do we fix this? (in OO)

- Using inheritance as before
  - –Override the fly() method in rubber duck as in quack()

Wait a minute

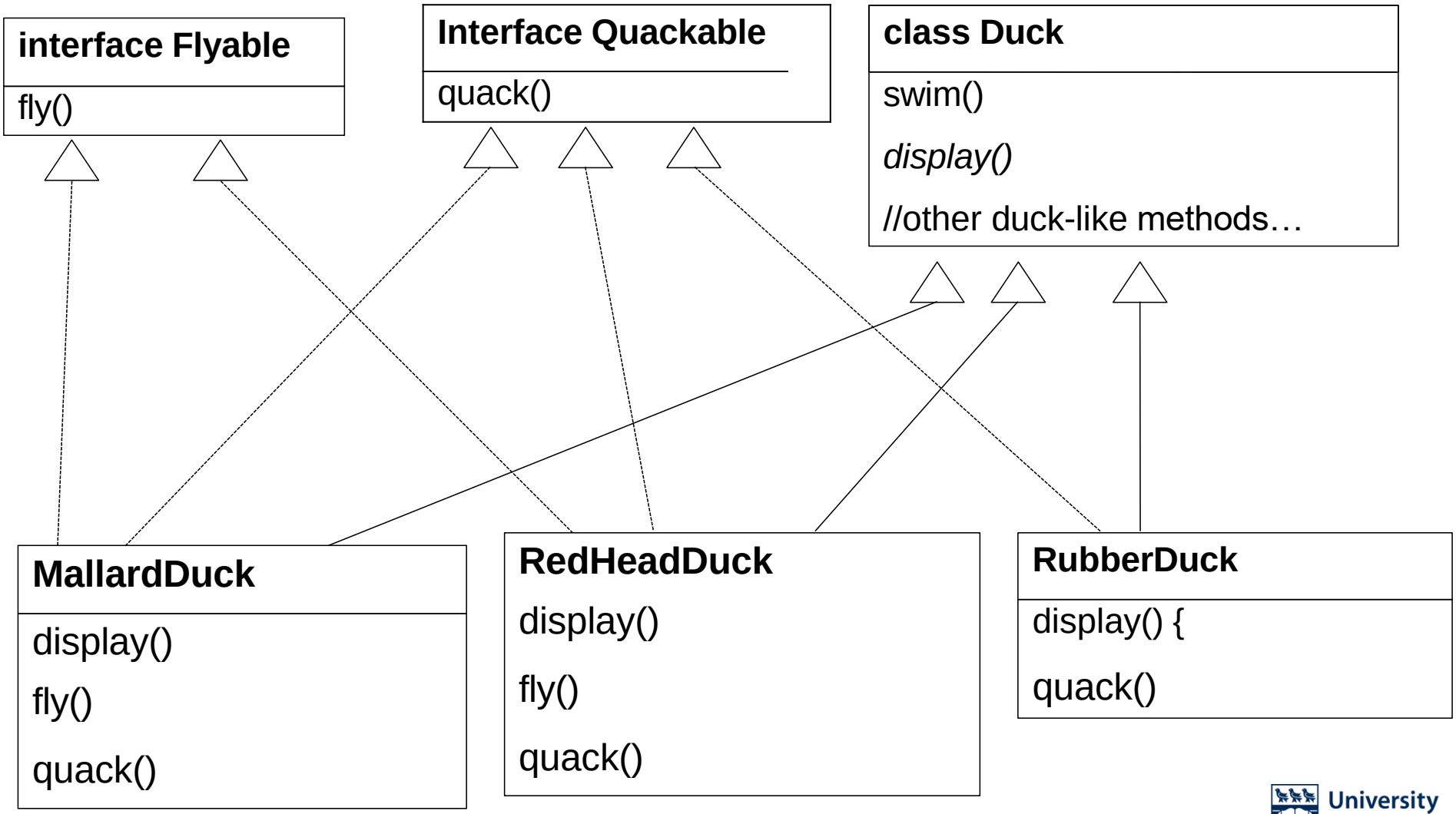
- How about new duck types?
  - –Decoy duck?
    - Can’t quack
    - Can’t fly

How else could we model the "variability"?

University of Victoria

How about Interface?

- Take the fly() method out of Duck superclass
- And make a Flyable() interface
  - –Only those ducks that fly are required to implement the interface
- Make a Quackable interface too



University of Victoria

But…

- You shoot yourself in the foot by duplicating code for every duck type that can fly and quack!
- And we have a lot of duck types
- We have to be careful about the properties – we cannot just call the methods blindly
- *We have created a maintenance nightmare!*

Re-thinking:

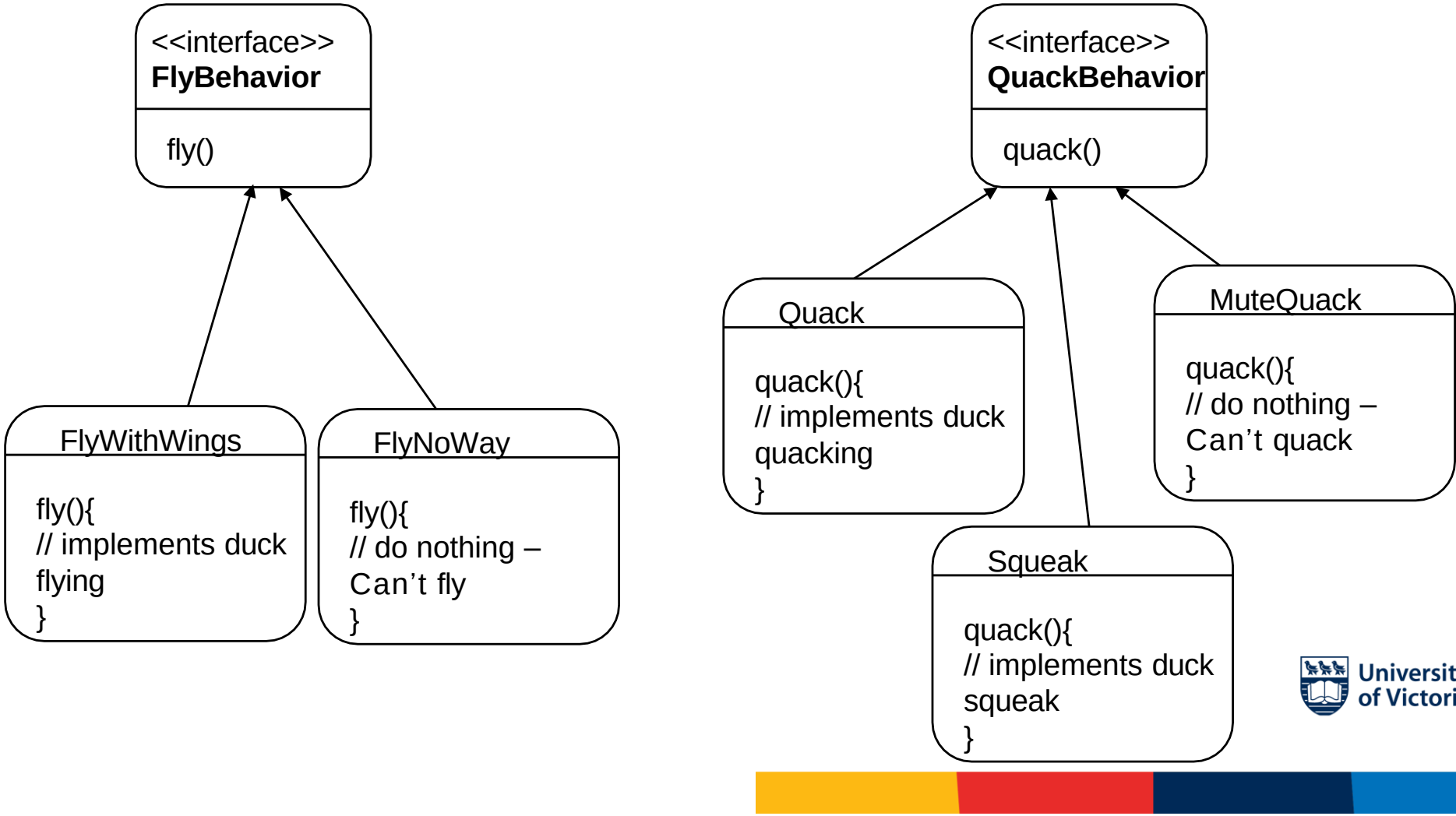
- Inheritance has not worked well because
  - Duck behaviour keeps changing
  - Not suitable for all subclasses to have those properties
- Interface was at first promising, *but*
  - No code re-use
  - Tedious
    - Every time a behaviour is changed, you must track down and change it in all the subclasses where it is defined
  - Error prone

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same

- So what are variable in the Duck class?
  - –Flying behaviour
  - –Quacking behaviour
- Pull these duck behaviours out of the Duck class
  - –Create new classes for these behaviours

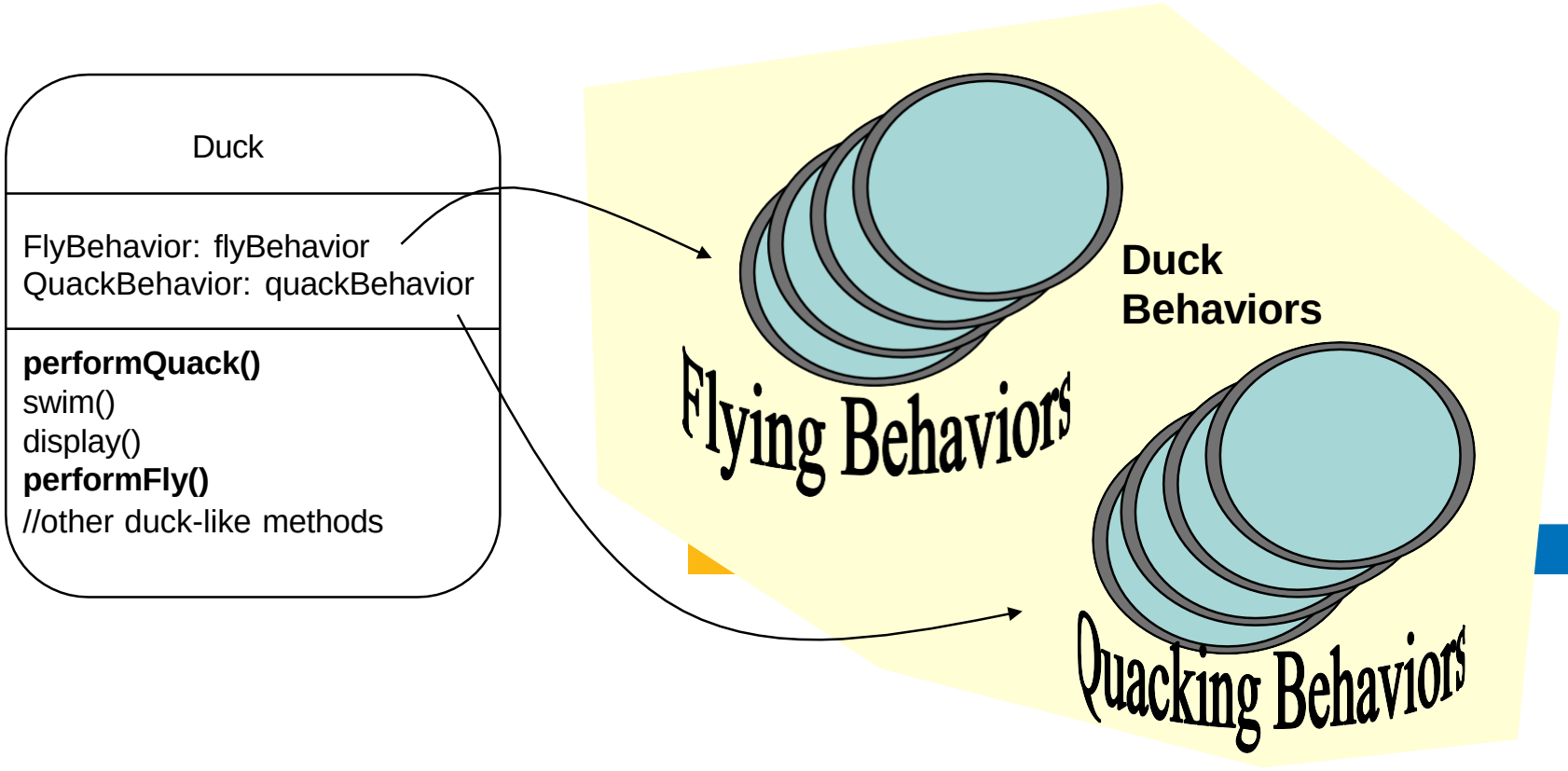
Implementing duck behaviours in separate strategy classes



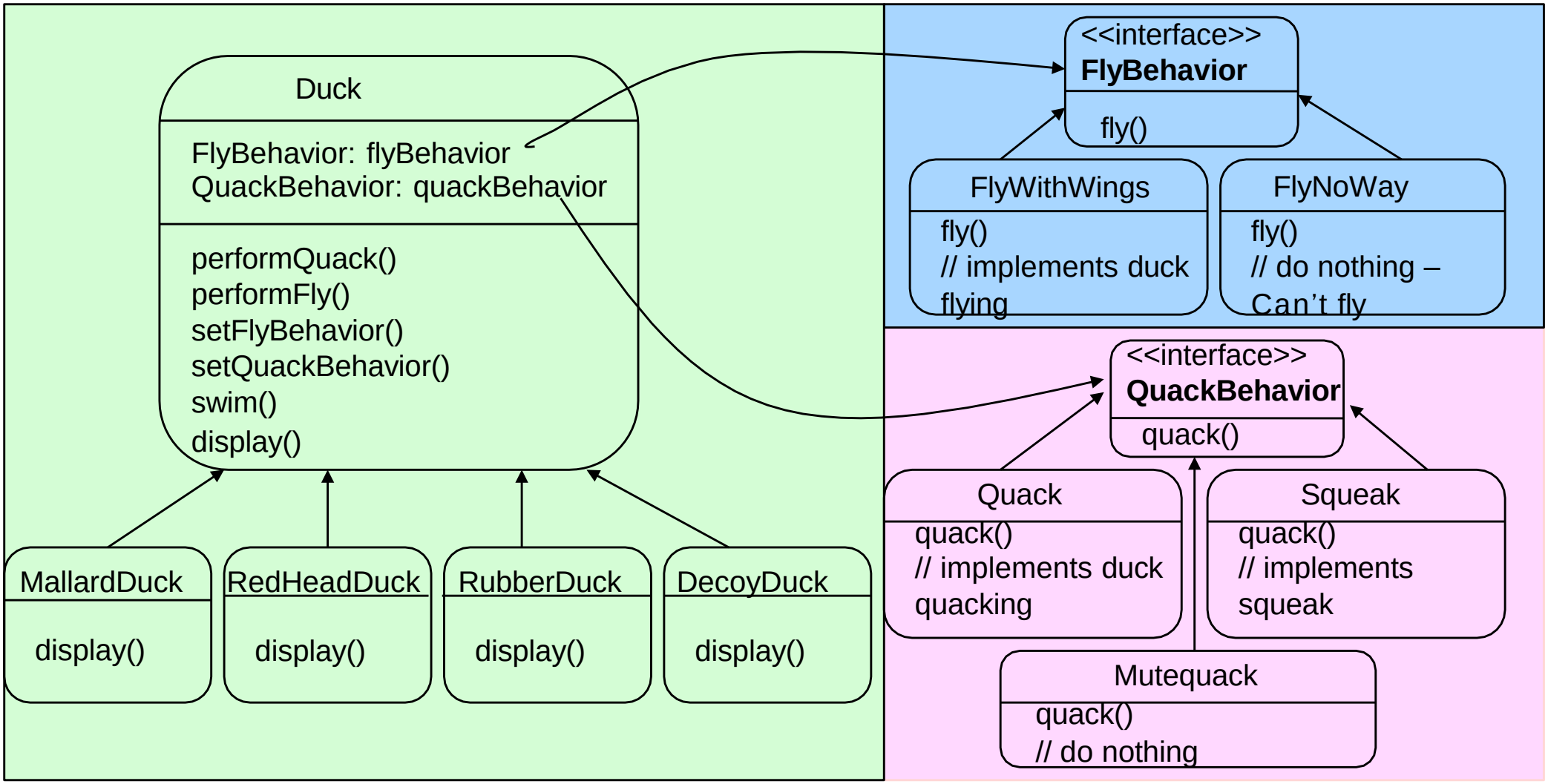
University of Victoria

Integrating the behaviours into the Duck class

- Duck class will ***delegate*** its flying and quacking behavior instead of implementing these itself

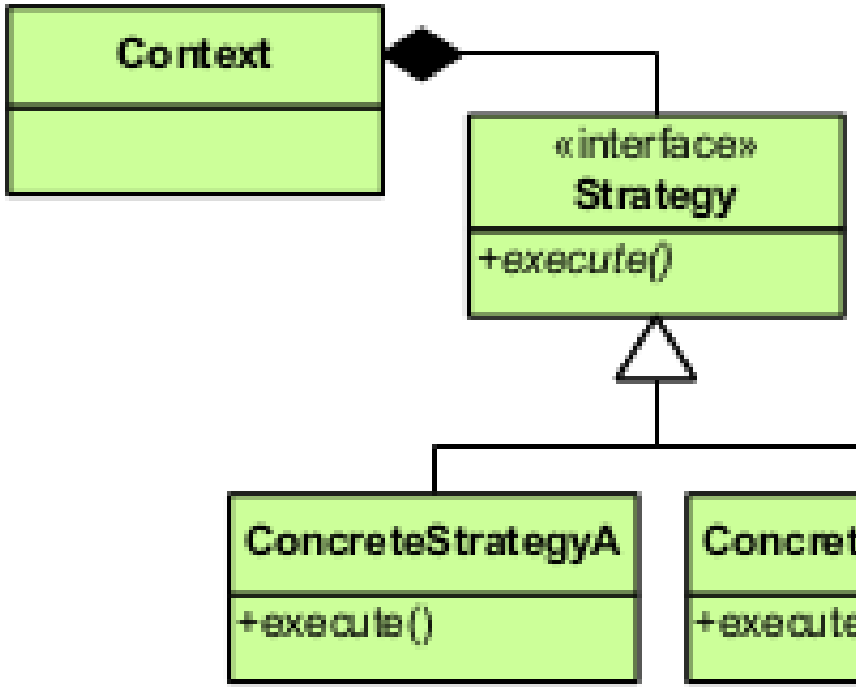


Using UML


Duck
FlyBehavior: flyBehavior
QuackBehavior: quackBehavior
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
swim()
display()
MallardDuck
display()
RedHeadDuck
display()
RubberDuck
display()
DecoyDuck
display()
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly()
// implements duck flying
FlyNoWay
fly()
// do nothing – Can't fly
<<interface>>
QuackBehavior
quack()
Quack
quack()
// implements duck quacking
Squeak
quack()
// implements squeak
Mutequack
quack()
// do nothing

Strategy Pattern Defined

The Strategy Pattern defines a family of algorithms, Encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Specific behaviours by implementing interface QuackBehavior

```
public class Quack implements QuackBehavior { public void quack() {
        System.out.println("Quack");
    }
}
```

```
public class Squeak implements QuackBehavior { public void quack() {
        System.out.println("Squeak");
    }
}
```

```
public class MuteQuack implements QuackBehavior { public void quack() {
        System.out.println("<< Silence >>");
    }
}
```


2. Implement performQuack()

```
public abstract class Duck {
    // Declare two reference variables for the behavior interface types
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior; // All duck subclasses inherit these
    // etc

    public Duck() {
    }


    public void performQuack() {
        quackBehavior.quack();   // Delegate to the behavior class
    }
```


3. How to set the quackBehavior variable & flyBehavior variable

```
public class MallardDuck extends Duck {


    public MallardDuck() {


      quackBehavior = new Quack();
                    // A MallardDuck uses the Quack class to handle its quack,
                    // so when performQuack is called, the responsibility for the quack
                    // is delegated to the Quack object and we get a real quack


      flyBehavior = new FlyWithWings();
                    // And it uses flyWithWings as its flyBehavior type

    }

    public void display() {
          System.out.println("I'm a real Mallard duck");
    }
```

How to set behaviour dynamically?

Add new methods to the Duck class

```
public void setFlyBehavior (FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```

Strategy Pattern in Game Dev (example)

https://youtu.be/MOEsKHqLiBM?feature=shared

In-class activity

- Find another “manifestation” of the Strategy pattern in the wild
- Submit to Week 5
- 15 minutes