

Design Patterns

Steven R. Bagley

Last Time

- Introduced the Strategy Pattern
- Saw how it could solve our aquatic problems...
- Two design principles

Design Principle

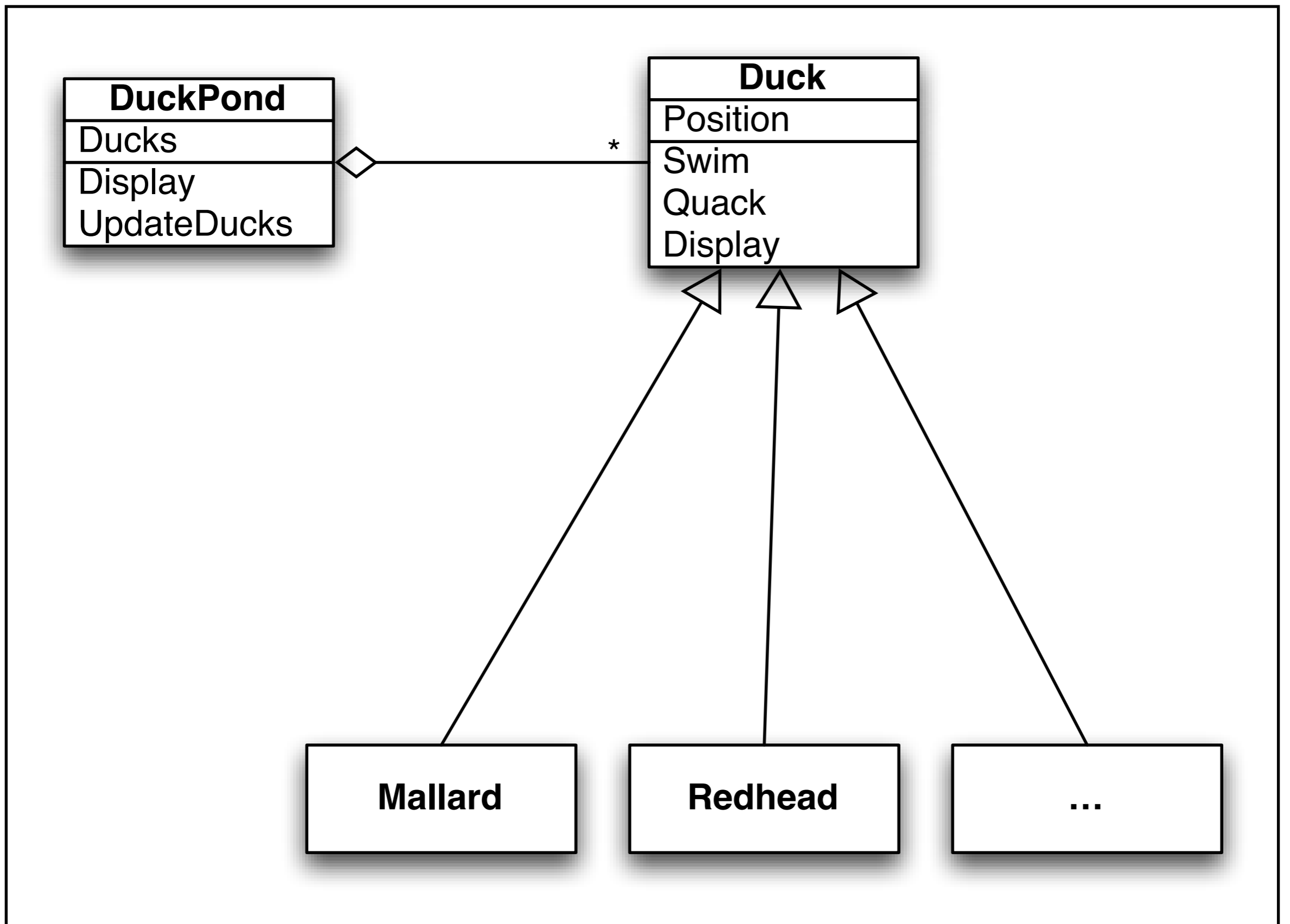
“Identify the aspects of your application that vary and separate them from what stays the same”

Design Principle

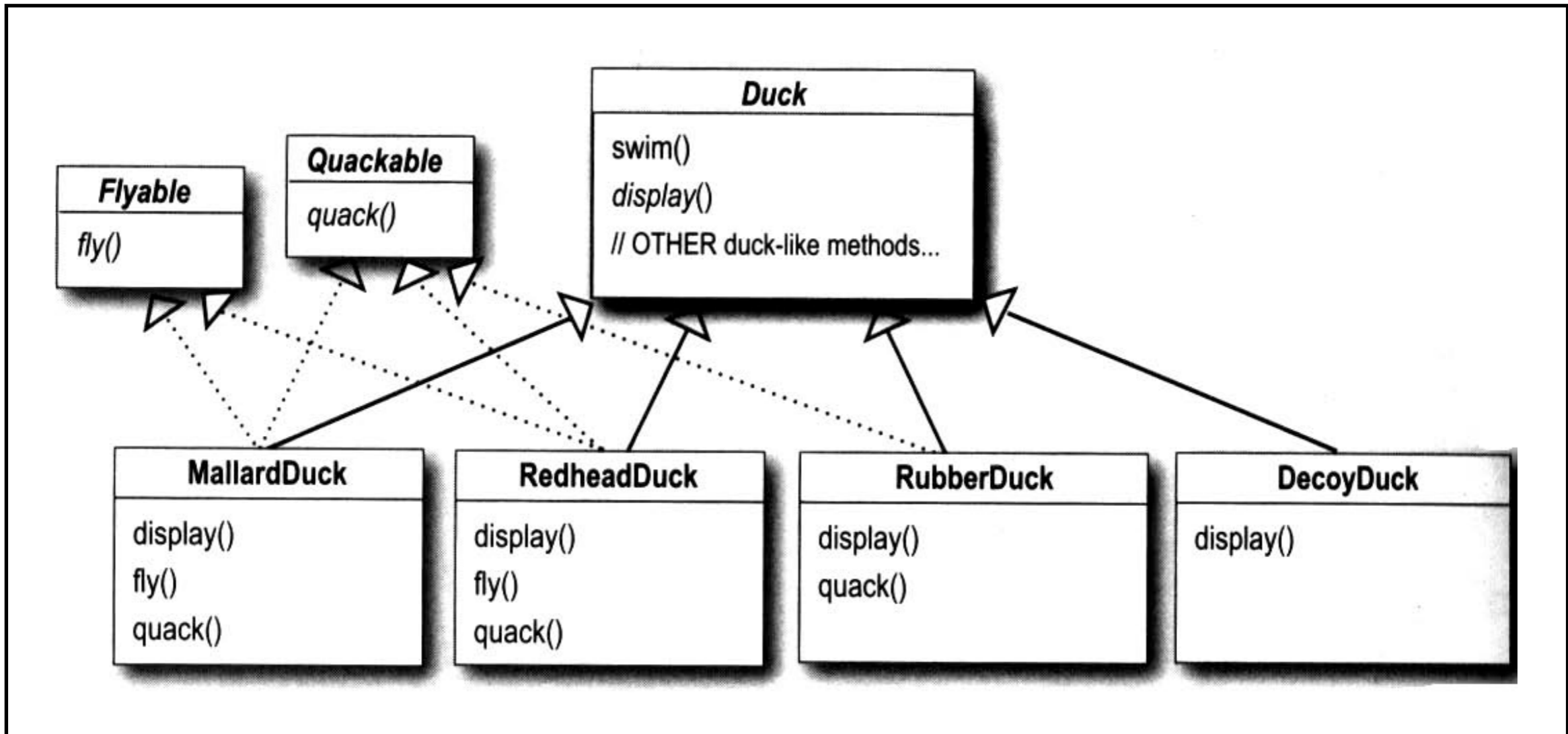
“Program to an interface,
not an implementation”

Today

- Consider the Strategy Pattern in more depth
- Design Patterns, the big picture



Worked fine until we tried to make them fly
when our rubber ducks started to fly
Lots of overriding methods to hide implementation



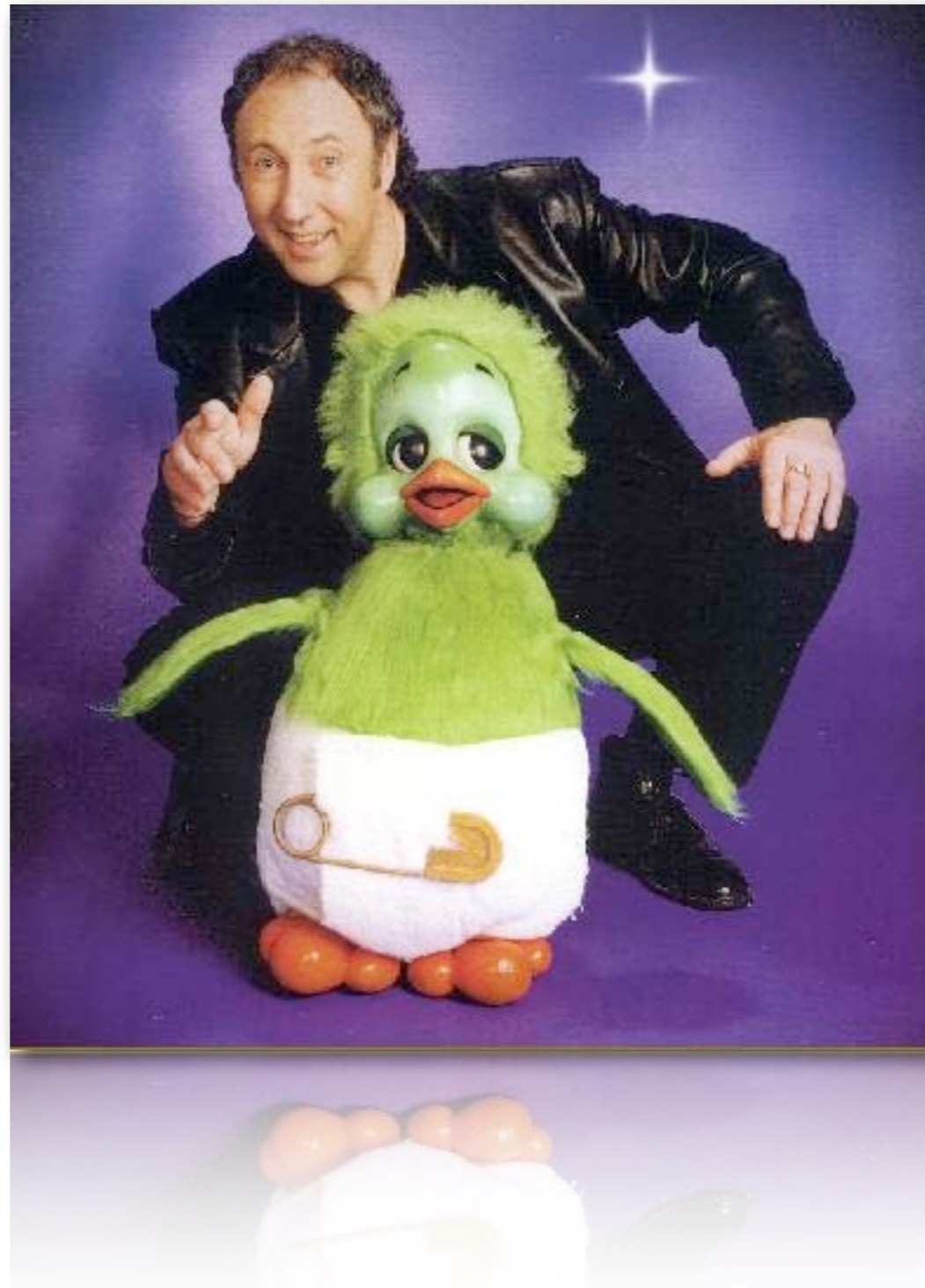
Strategy Pattern

- 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
- FlyBehaviour, QuackBehaviour
- Loosely coupled

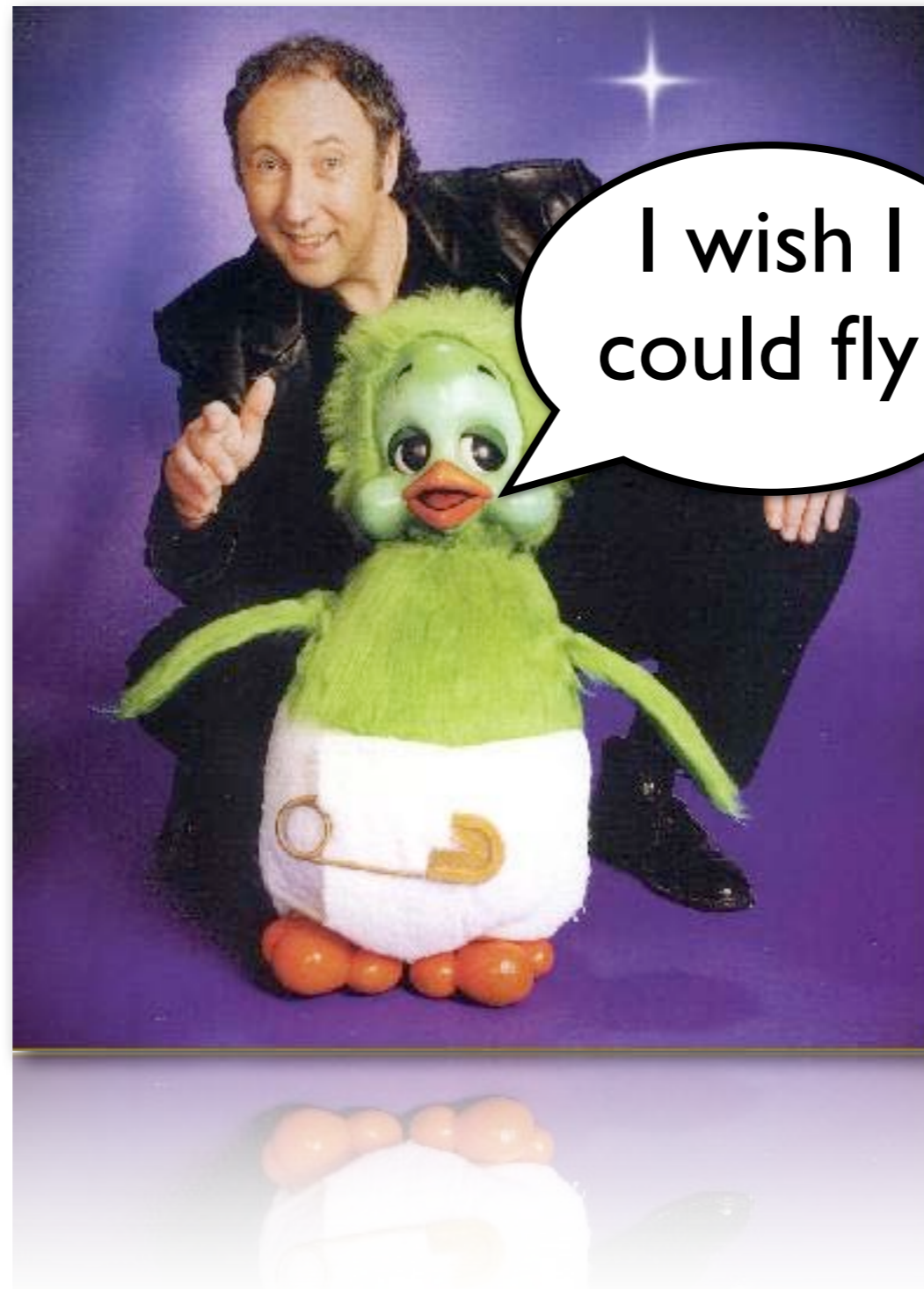
```
class CDuck
{
public:
    void PerformQuack();
    virtual void Swim();
    virtual void Display() = 0;
    void PerformFly();

protected:
    FlyBehaviour *m_flyBehaviour;
    QuackBehaviour *m_quackBehaviour;
};
```

Change in Strategy



Change in Strategy



Change in Strategy



Orville

- Orville is a Duck
 - That can't fly
 - Sings, not quack
- Can easily implement him
- Need a new behaviour
`QuacksWithIrritatingSong`

```
#include "Duck.h"

class CMallard : public CDuck
{
public:
    CMallard();

    virtual void Display();
};
```

```
#include "Duck.h"

class COrville : public CDuck
{
public:
    COrville();

    virtual void Display();
};
```

```
CMallard::CMallard()
{
    m_flyBehaviour = new CFlyWithWings;
    m_quackBehaviour = new CQuack();
}

void CMallard::Display()
{
    printf("I'm a real Mallard duck\n");
}
```

```
COrville::COrville()
{
    m_flyBehaviour = new CFlyNoWay;
    m_quackBehaviour = new CQuacksSillySong();
}

void COrville::Display()
{
    printf("I'm Orville..\n");
}
```

Orville

- Representation of new ducks is trivial
- Subclass Duck
- Add relevant behaviours
- Change Behaviour at runtime

Behaviour Modification

- Really Simple
- Mutators on `Duck` class
- `flyBehaviour` replaced with new instance
- Client can use Mutator to change behaviour

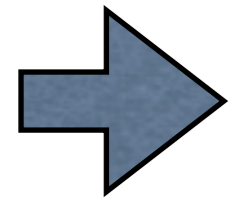
```
class CDuck
{
public:
    void PerformQuack();
    void PerformFly();
    void SetFly(FlyBehaviour *fb);
    void SetQuack(QuackBehaviour *qb);
    virtual void Swim();
    virtual void Display() = 0;

protected:
    FlyBehaviour *m_flyBehaviour;
    QuackBehaviour *m_quackBehaviour;
};
```

```
void CDuck::SetFly(FlyBehaviour *fb)
{
    if(fb)
    {
        if(m_flyBehaviour)
            delete m_flyBehaviour;

        m_flyBehaviour = fb;
    }
}
```

```
COrville *orville = new COrville();  
  
orville->Display();  
orville->performFly();  
orville->SetFly(new CFlyWithRocket);  
orville->performFly();
```



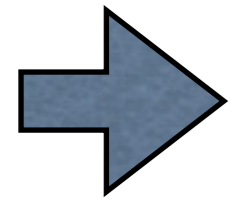
```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```



```
COrville *orville = new COrville();
```

```
orville->Display();
```

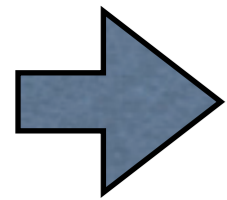
```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```

| | |
|------------------|------------------|
| m_flyBehaviour | CFlyNoWay |
| m_quackBehaviour | CQuacksSillySong |

```
COrville *orville = new COrville();
```



```
orville->Display();
```

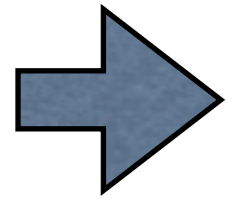
```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```

| | |
|------------------|------------------|
| m_flyBehaviour | CFlyNoWay |
| m_quackBehaviour | CQuacksSillySong |

```
COrville *orville = new COrville();
```



```
orville->Display();
```

```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```

| | |
|------------------|------------------|
| m_flyBehaviour | CFlyNoWay |
| m_quackBehaviour | CQuacksSillySong |

I'm Orville..

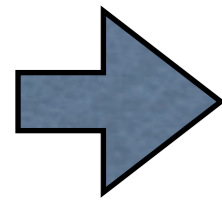
```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```



| | |
|------------------|------------------|
| m_flyBehaviour | CFlyNoWay |
| m_quackBehaviour | CQuacksSillySong |

I'm Orville..

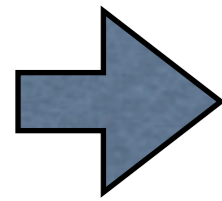
```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```



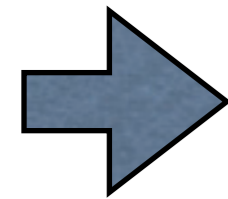
| | |
|------------------|------------------|
| m_flyBehaviour | CFlyNoWay |
| m_quackBehaviour | CQuacksSillySong |

Can't Fly

```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```



```
orville->SetFly(new CFlyWithRocket);
```

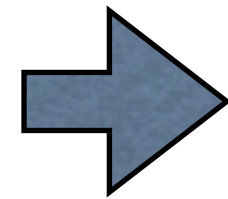
```
orville->performFly();
```

| | |
|------------------|------------------|
| m_flyBehaviour | CFlyNoWay |
| m_quackBehaviour | CQuacksSillySong |

```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```



```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```

| | |
|------------------|------------------|
| m_flyBehaviour | CFlyWithRocket |
| m_quackBehaviour | CQuacksSillySong |

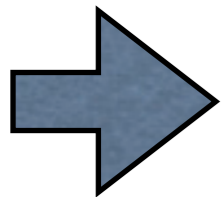
```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```



| | |
|------------------|------------------|
| m_flyBehaviour | CFlyWithRocket |
| m_quackBehaviour | CQuacksSillySong |

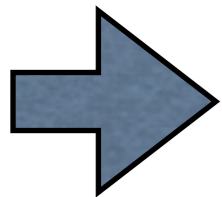
```
COrville *orville = new COrville();
```

```
orville->Display();
```

```
orville->performFly();
```

```
orville->SetFly(new CFlyWithRocket);
```

```
orville->performFly();
```



| | |
|------------------|------------------|
| m_flyBehaviour | CFlyWithRocket |
| m_quackBehaviour | CQuacksSillySong |

Behaviour Modification

- Inheritance locks the behaviour into the class at compile-time
- Cannot be changed
- Except by copying the contents of the object into a new one of a different class
- Strategy Pattern (using composition) can be changed at runtime

Design Principle

“Favour composition over inheritance”

Has-a better than is-a

- Has-a relationship means it can be changed, is-a means fixed
- Many Design Patterns use composition over inheritance
- Our behaviours can be used elsewhere

Intrinsic/Extrinsic

- Use Inheritance for intrinsic properties
- Use Composition for extrinsic properties

Strategic Composition

- Our objects can be set to *any* behaviour
- Default behaviour, that can be overridden
- Provided it implements the correct interface
- Even if they are implemented later
- Supports Change

Strategy Pattern

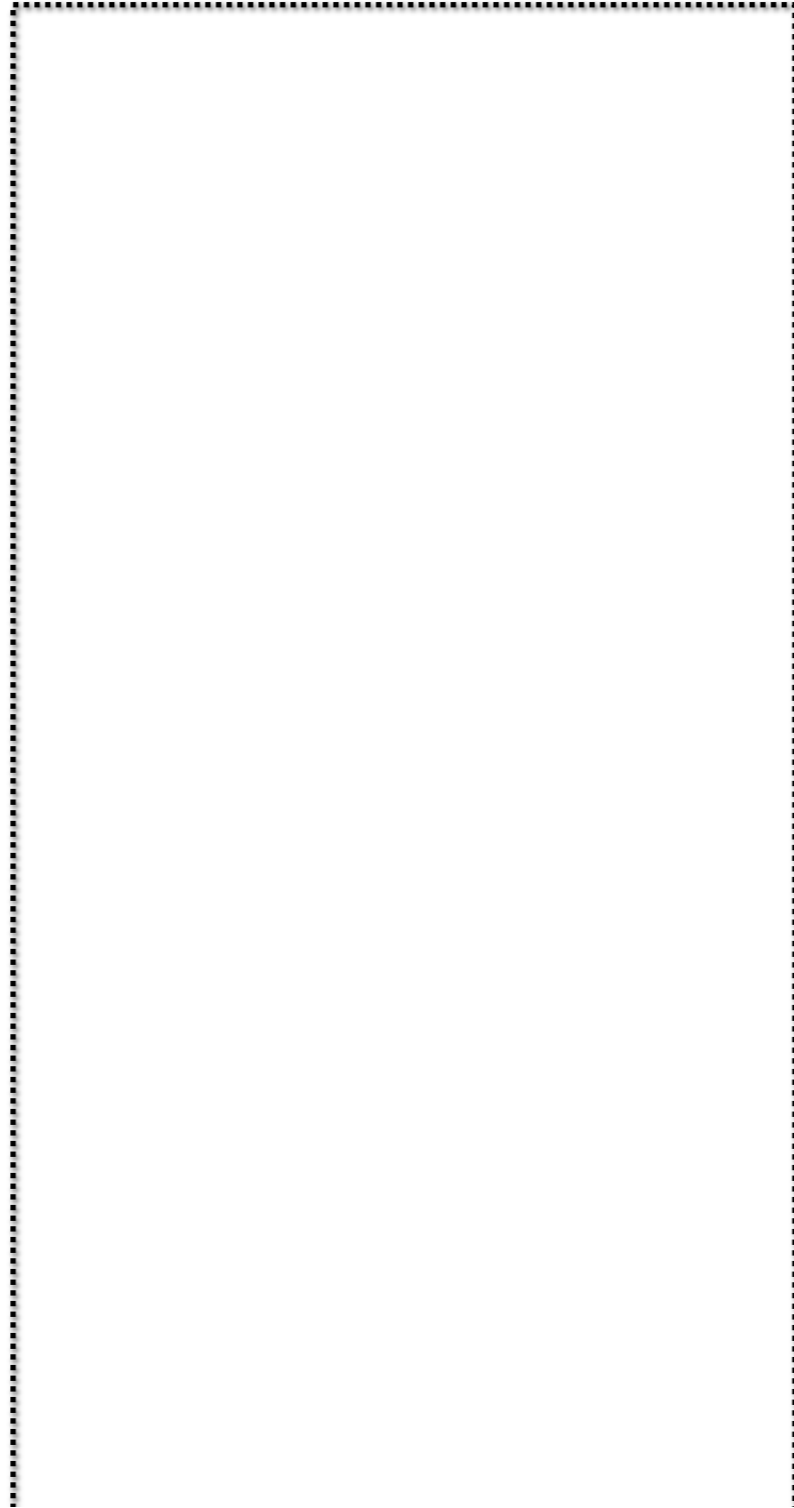
- Duck behaviours are very simple
- Do not modify the `Duck` Object
- Some algorithms may need to work on the data inside the object
- Won't this break encapsulation?

Line Breaking

- Formatting blocks of text
- Text stores as a long run of characters
- Various algorithms break it into lines
- Line-breaking algorithm may vary

Jackdaws love my big sphinx of quartz

Jackdaws love my big sphinx of quartz



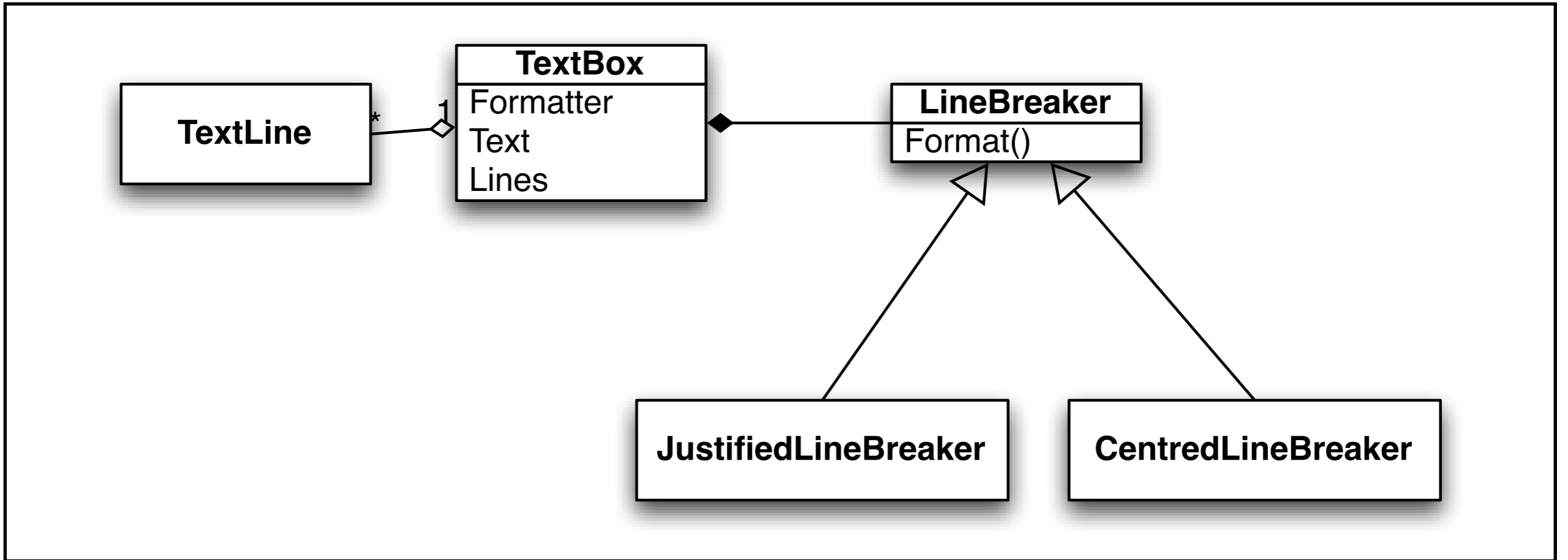
Jackdaws love my big sphinx of quartz

Jackdaws love
my big sphinx of quartz

Jackdaws love
my big sphinx
of quartz

Line-breaking Strategy

- Classic Case for Strategy Pattern
- Can't vary algorithm with inheritance
- Separate `TextBox` from formatting algorithm
- Formatting goes in a `LineBreaker` object
- But `TextBox` contains the lines after formatting...



Break Encapsulation

- Line breaking algorithm needs to modify internals of `TextBox`
- How do we do this?
- Make the data public?
- NOOOOOOOOOOOO!!!!!!!

Make Friends?

- What about making classes be friends?
- A class 'friend' is a concept that enables one class to specify that another class can modify its internals
- Supported in C++ and C# but not Java
- Better than public data

Break Friends

- Friends are not a good idea
- Line-breaking objects would be tied to working on `TextBox` only
- If `TextBox` changes, Line Breakers must be changed
- Tight coupling

Design Principle

“Program to an interface,
not an implementation”

Line-break interface

- Separating the algorithm from the data
- Loose coupling
- `LineBreaker` should be distinct from `TextBox` (might want to use it elsewhere)
- Define an interface that `LineBreaker` can use to modify `TextBox`

LineBreaker Interface

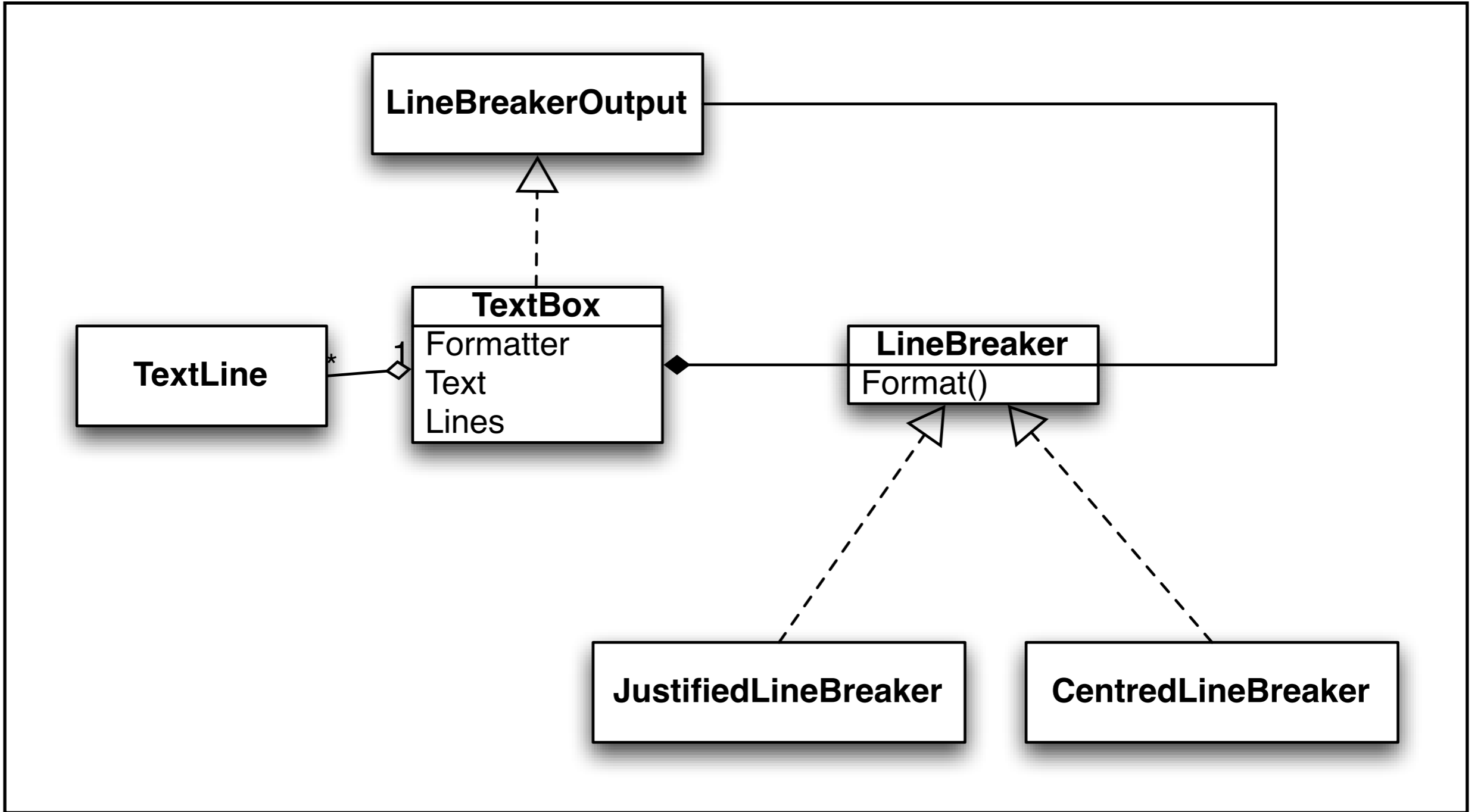
- What does `LineBreaker` need to do?
 - Get the text to format?
 - Break into lines
 - Pass the lines of text back to the `TextBox` object
- This could be an interface implemented by `TextBox`

Fitting it together

- `TextBox` has-a `LineBreaker`
- `TextBox` calls `format()` on `LineBreaker`
- `format` takes an interface as a parameter
- `LineBreaker` uses interface to access text
- `LineBreaker` unaware of `TextBox`

What interface?

- `TextBox` implements the interface `LineBreaker` uses
- `TextBox` has full control on how it uses the output of the algorithm
- `TextBox` still responsible for internal data, encapsulation intact (phew!)



Using Patterns

- Design Patterns are for us, not the computer
- Start to think about problems at the pattern level, not the object level
- But don't get *pattern fever* (patterns are not good for Hello World)
- Shared vocabulary for a development team

Patterns communicate a lot of information to other developers, quickly and simply in less words. The other developer knows exactly the kind of design you intend but without you needing to outline every object's job. Keeps you focused on the design, without getting bogged down in implementation details (think about the ducks, Strategy solved problems that appeared in implementation)

Patterns for the Brain

- Unlike Libraries/Frameworks, Patterns are for you
- Can't build a class library since they are higher level
- Think about a bridge
 - Each Bridge design is specific
 - But the design follow the same pattern

Isn't it just OO design?

- OO principles don't guarantee a good design
- Hard to come up with a good design straightaway
- Patterns provide proven designs that we can reuse
- Gathered into catalogues

Again think of bridges

Pattern not found

- But what happens if you can't find a pattern that fits
- Use the design principles that underly the patterns
- Think about how the design might need to change in the future

OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

OO Principles

- Encapsulate what varies
- Favour composition over inheritance
- Program to interfaces, not implementations