Design Patterns

Design challenges

- **Requirement** changes is the number one challenge
- The design **process** is challenging because essential design process is often done in an ad-hoc manner.
- **Technology** changes fast and ever-changing.
- Designers are constantly faced with numerous pressures from stakeholders.
  - Competition and time to market
- Can designs be described, codified or standardized?
  - this would short circuit the trial and error phase
  - produce "better" software faster

Design Pattern

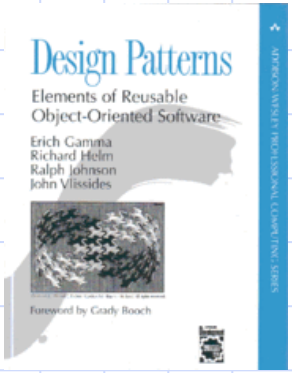
- What is Design Pattern
  - Design patterns represent the best practices used by experienced object-oriented software developers.
    - Solutions to general problems that software developers faced during software development.
    - Solutions obtained by trial and error by numerous software developers over long time

History of patterns

- the concept of a "pattern" was first expressed
  in Christopher Alexander's work *A Pattern Language* in 1977 (2543 patterns)

- in 1990 a group called the Gang of Four or "GoF"
  (Gamma, Helm, Johnson, Vlissides) compiled a catalog of design patterns

- *Design Patterns Book 1995: Elements of Reusable Object-Oriented Software*



Benefits of patterns

- Why to Reinvent the Wheel?
  - If someone has already solved a problem why shouldn't use his solution as a pattern?
  - Learning these patterns helps inexperienced developers to learn software design in an easy and faster way
- Patterns provide a common vocabulary
  - allows engineers to abstract a problem
  - Allows engineers to talk about an abstract solution in isolation from its implementation
  - promotes design reuse and avoid mistakes
  - improves documentation (may be less documentation is needed).

Several Type of Design Patterns

- Creational Patterns

  These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using the constructor

- Structural Patterns

  These design patterns concern class and object composition.

- Behavioral Patterns

  These design patterns are specifically concerned with communication between objects.

Gang of Four (GoF) patterns

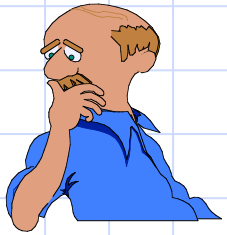
- **Examples of Creational Patterns**
  (abstracting the object-instantiation process)
  - Factory Method
  - Builder
  - Abstract Factory
  - Singleton
  - Prototype
- **Examples Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - Adapter
  - Bridge
  - *Composite*
  - *Decorator*
  - Façade
- **Examples of Behavioral Patterns**
  (communication between objects)
  - Command
  - *Iterator*
  - Mediator
  - *Observer*
  - *Strategy*

Common Practices

- Issue:
  - "CHANGE" is the main challenge in software development lifecycle.
- Concern:
  - The biggest concern is how to minimize or remove the impact of change.
- Design Principle:
  - Identify the aspects of the application that vary and program them to an interface, not an implementation.

Common Design Issues Walkthrough

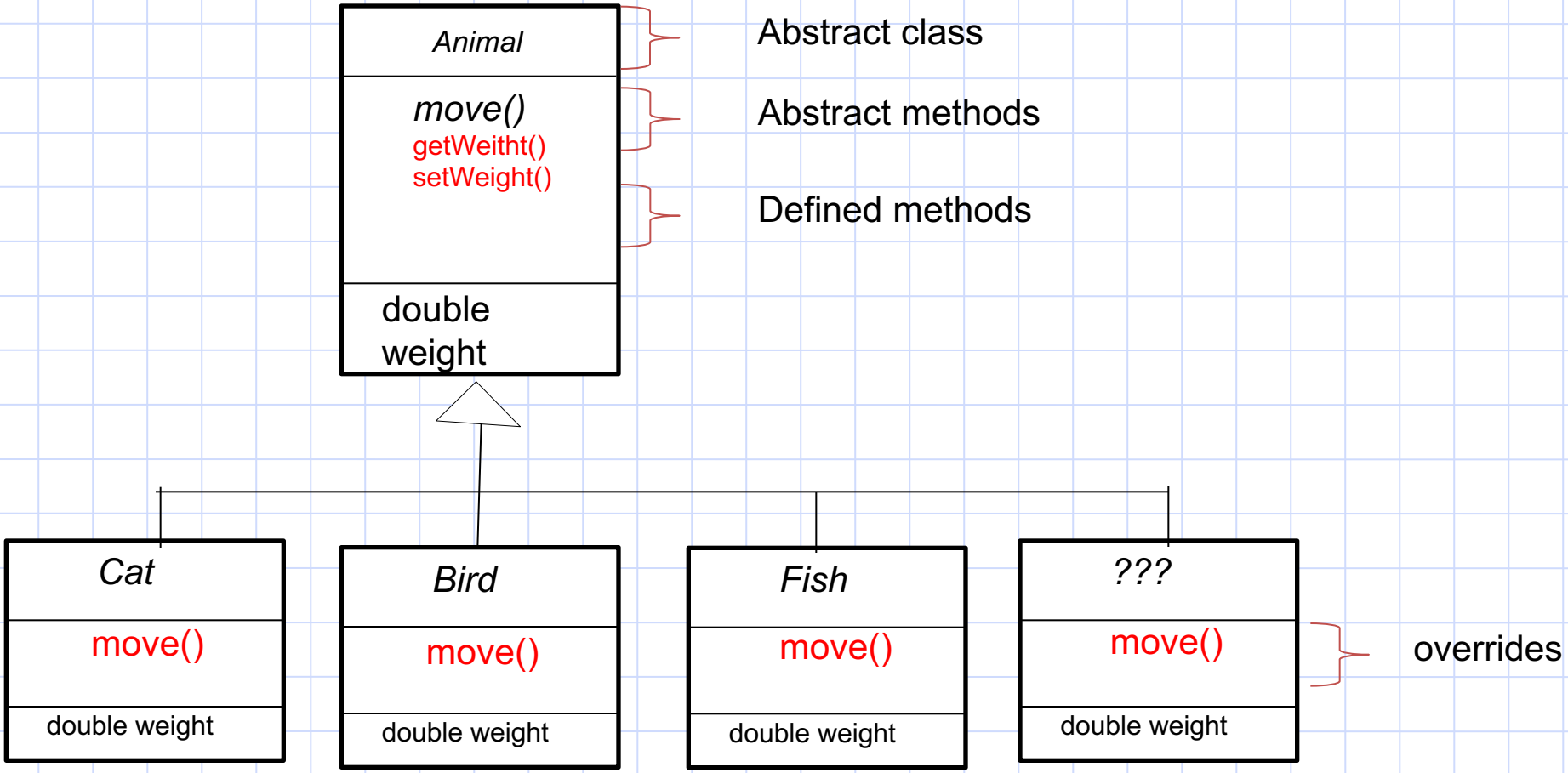
Discussions



- Lets assume you have been assigned as team of engineers to design a game for kids that need many animals to be created and to be able to move and make sounds …

Example

- Here is a possible partial class design for such an application:



And, here is a possible Implementation

```
abstract class Animal {
        abstract public void move();
        public double getWeight() {
                return weight;
        }
        public void setWeight(double weight) {
                this.weight = weight;
        }
        private double weight;
}

class Cat extends Animal {
        public void move() {
                System.out.println("Walking");
        }
}
```

```
class Bird extends Animal {
        public void move() {
        System.out.println("Flying");
        }
}

class Fish extends Animal {
        public void move() {
                System.out.println("Swimming");
        }
}

class Cricket extends Animal {
        public void move() {
                System.out.println("Jumping");
        }
}
```

- But the issue is the that the "change" is a constant need in software development.

http://www.hrmonline.ca/blog/alana-free/how-hr-can-prepare-for-the-four-phases-of-change-225680.aspx

- Lets assume we have decided to add sound or talk behavior for all animals.

Implementation with the required changes

```
abstract class Animal {
        abstract public void move();
        abstract public void talk();

        public double getWeight() {
                return weight;
        }
        public void setWeight(double weight) {
                this.weight = weight;
        }
        private double weight;
}

class Cat extends Animal {
        public void move() {
                System.out.println("Walking");
        }
        public void talk() {
                System.out.println("Meowing");
        }
}
```

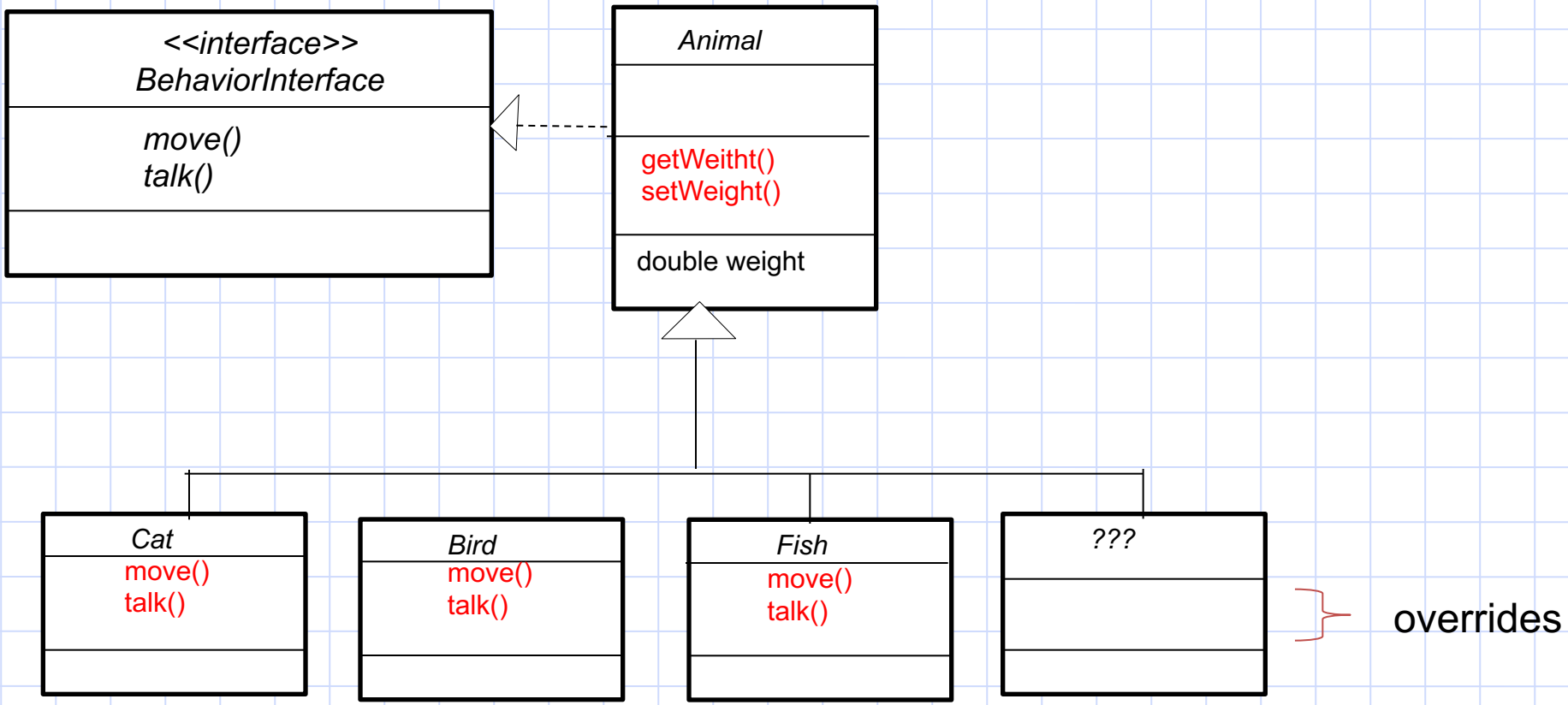
```
class Bird extends Animal {
        public void move() {
                System.out.println("Flying");
        }
        public void talk() {
                System.out.println("Tweeting");
        }
}

class Fish extends Animal {
        public void move() {
                System.out.println("Swimming");
        }
        public void talk() {
                System.out.println("No sound");
        }

}
```

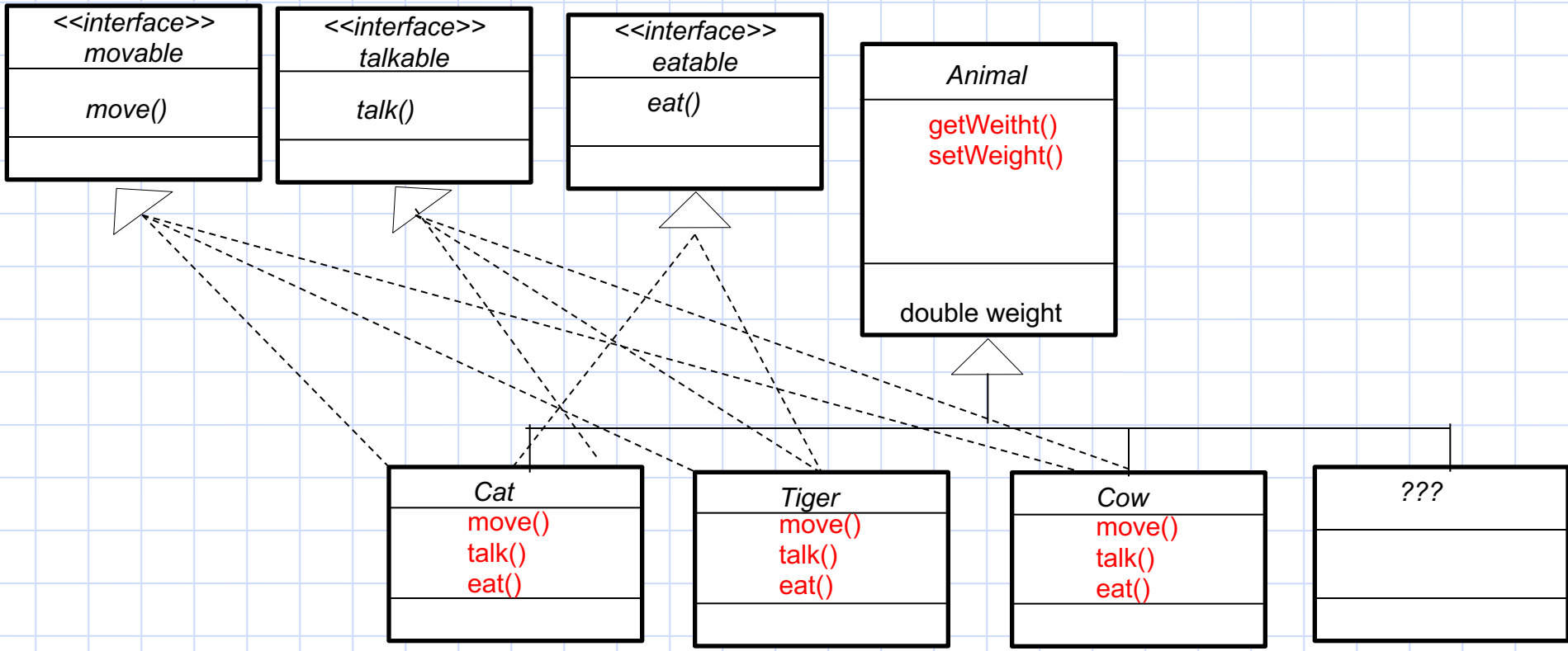
This solution requires too many change

Would This Solution Help?



Not really, still the same issue. You need to make changes to all descending classes.

What About This One?



This is even more complicated and not a better solution.
A maintenance nightmare

- Lets even look at other possible issues. What if we need to dynamically (at the runtime) to change the ability of an animal. What if at some point we want to come up with a type fish that can walk. Don't forget that, in the virtual world everything is possible.
- Is there a be better solution?

A Better Approach

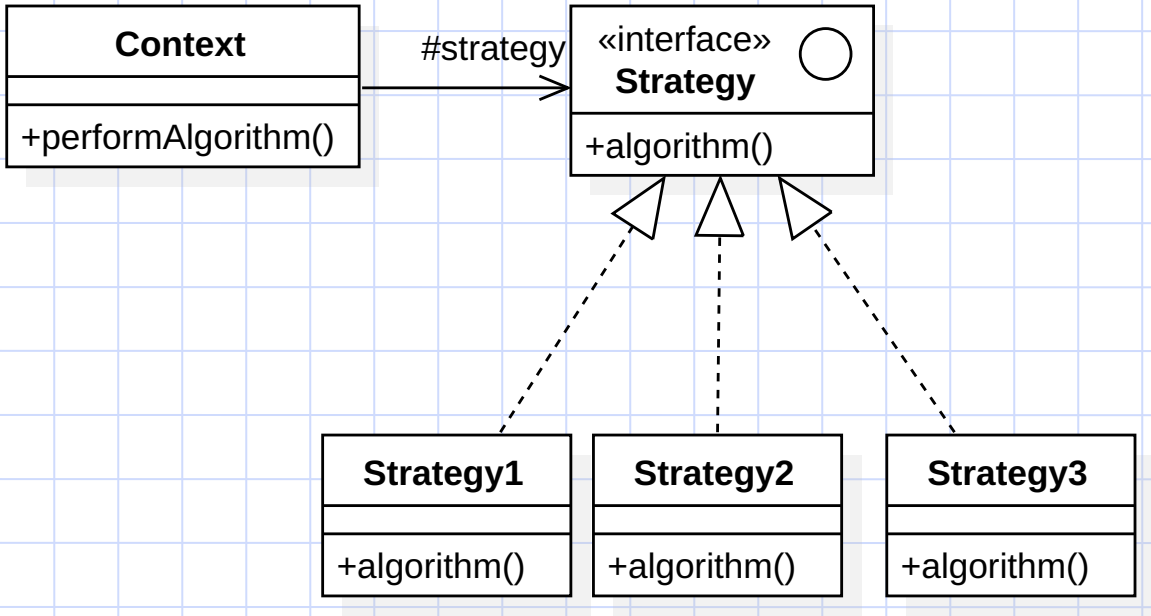
1. Identify the aspects/behaviors of the application that are subject to the changes from those that stay unchanged.
2. Create an interface for each changeable behavior.
3. For each interface, create a class that only implements that interface.

Strategy Pattern

Fromhttps://www.linkedin.com/pulse/strategy-execution-john-depalma

Strategy Pattern Model

- Definition of Strategy Pattern
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Strategy pattern

- **strategy**: Algorithm(s) separated from a class, and **encapsulated as separate class**.
- each strategy implements **one behavior**.
- allows changing an object's behavior dynamically without extending / changing the object itself

- examples:
  - file saving/compression algorithms:
  - layout managers on GUI containers
  - AI algorithms for computer game players

How Can We Implement Strategy Pattern

- General Format in Java:
  1) Find out which functionalities are subject to change in future
  2) Design a strategy interface
  3) Have as many as needed strategy classes to implement the interface design in (2). All functionalities that considered in (1) should appear as class that implements interface in (2)
  4) Make sure your core classes having a reference of type strategy interface.
  5) Make sure your core classes have methods to set the strategy, and all strategy method from strategy classes, as needed.

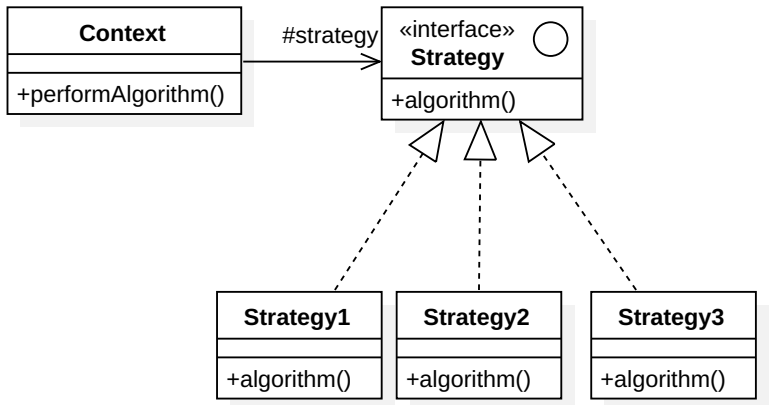
How these steps can be implemented in Java and C++?

A General and Simple Template for Strategy Pattern in Java

```
Interface Strategy{
  public void doSomething();
}
class Strategy1 implements Strategy{
 public void doSomething() {
     // implementation
     // implementation second algorithm

  }
 }
class Strategy2 implements Strategy{
public void doSomething() {
   // implementation second algorithm
  }
 }
 // MORE STRATEGIES CAN GO HERE
```

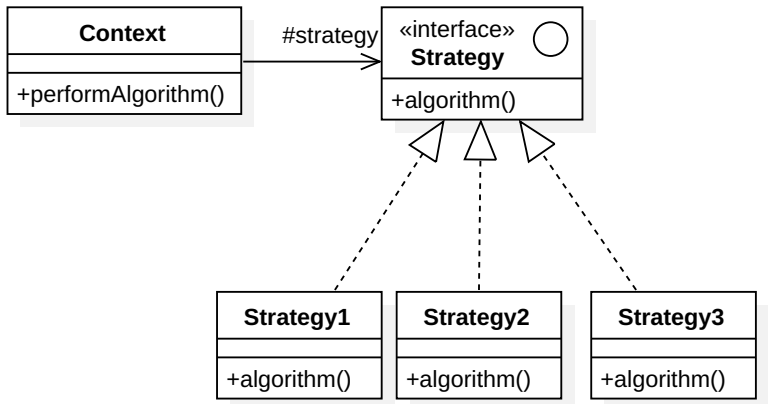
```
class Context {
   Strategy str;
  public Context() {
      // str can be set to Strategy1 by default
      // code to initialize data members
  }
  public void setStrategy1(Strategy s) {
   // s can be Strategy1 or Strategy2
    str = s;
  }
  public void performStrategy1() {
    str.doSomething();
  }
}
```



A General and Simple Template for Strategy Pattern in C++

```
class Strategy{
public:
  virtual void doSomething() = 0;
};
class Strategy1: public Strategy{
public:
  void doSomething() {
     // implementation
     // implementation second algorithm
  }
 };
class Strategy2: public Strategy{
public:
void doSomething() {
   // implementation second algorithm
  }
 };
// MORE STRATEGIES CAN GO HERE
```

```
class Context {
   Strategy* str;
  Context() {
    //optionally  str can be set to Strategy1 by default
    // code to initialize data members
  }
  void setStrategy1(Strategy& s) {
    // s can be Strategy1 or Strategy2
     str = &s;
  }
  void performStrategy1()const {
     str.doSomething();
  }
};
```



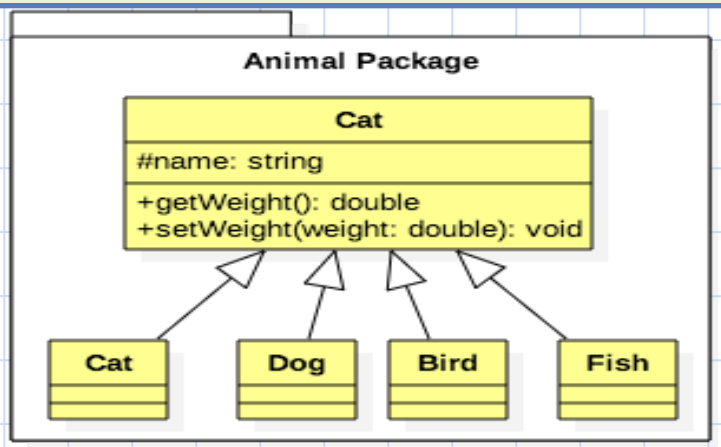
Strategy Pattern Example

Let's Try It

- Assume you are developing a game for children and you have designed a set of core classes such as Animal, Cat, Dog, fish, etc. Now you need to add functionalities such as walk, swim, etc.
- However the issues is that in this game for children what are your future requirements. You may need a Cat not only to be able to walk but also to be able to swim. Or who knows in this game you may want to allow like many cartoons that fish also walks.

```
class Animal {
        public Animal(double wi){
                weight = wi;
                name = "";
        }
        public double getWeight() {
                return weight;
        }
        public void setWeight(double weight) {
                this.weight = weight;
        }
        protected double weight;
        protected String name;
};
```

```
class Cat extends Animal {
        public Cat(double w) {
                super(w);
                name = "Cat";
        }
}
//-------------------------------------------------------
class Dog extends Animal {
        public Dog(double w) {
                super(w);
                name = "Dog";
        }
}
//-------------------------------------------------------
class Bird extends Animal {
        public Bird(double w) {
                super(w);
                name = "Bird";
        }
}
//-------------------------------------------------------
class Fish extends Animal {
        public Fish(double w) {
                super(w);
                name = "Fish";
        }
}
```



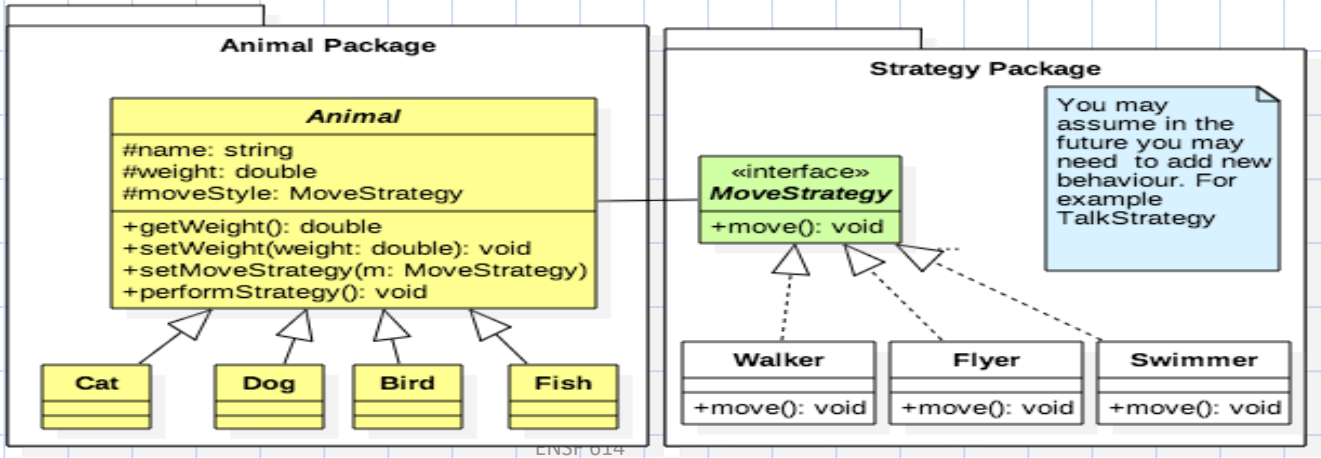
Example: Implementing Strategy Pattern in Java

```
interface MoveStrategy {
        abstract void move(String s);
}

//-------------------------------------------------------
class Walker implements MoveStrategy {
  public void move(String s) {
        System.out.println("Walking" + " " + s );
  }
}
//-------------------------------------------------------
class Flyer implements MoveStrategy {
  public void move(String s) {
        System.out.println("Flying" + " " + s);
  }
}

class Swimmer implements MoveStrategy {
  public void move(String s) {
        System.out.println("Swimming" + " " + s);
 }
}
```

```
abstract class Animal {
        public Animal(double wi){
                weight = wi;
                name = "";
        }
        public double getWeight() {
                return weight;
        }
        public void setWeight(double weight) {
                this.weight = weight;
        }
        public void setMoveStrategy(MoveStrategy m) {
                moveStyle = m;
        }
        public void performMove() {
                moveStyle.move(name);
        }
        protected double weight;
        protected String name;
        protected MoveStrategy moveStyle;
}
```



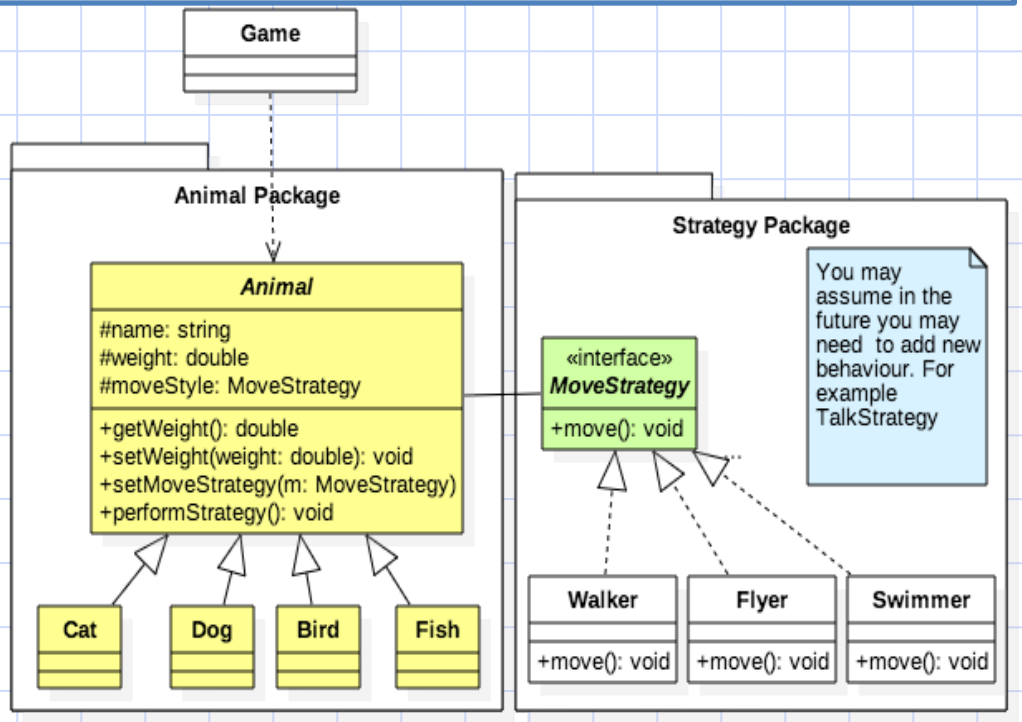
Example: Implementing Strategy (cont'd)

```
class Cat extends Animal {
        public Cat(double w) {
                super(w);
                    name = "Cat";
                moveStyle = new Walker ();
        }
}
//-------------------------------------------------------
class Dog extends Animal {
        public Dog(double w) {
                super(w);
                name = "Dog";
                moveStyle = new Walker ();
        }
}
//-------------------------------------------------------
class Bird extends Animal {
        public Bird(double w) {
                super(w);
                name = "Bird";
                moveStyle = new Flyer ();
        }
}
//-------------------------------------------------------
class Fish extends Animal {
        public Fish(double w) {
                super(w);
                name = "Fish";
                moveStyle = new Swimmer ();
        }
}
```

```
public class Game
{
  public static void main (String [] s) {
  Cat kitty = new Cat(100);
  //prints- Walking Cat
  kitty.performMove();

  Bird birdy = new Bird(400);
  //prints Flying Bird
  birdy.performMove();

  Fish fishy = new Fish(20);
  // prints Swimming Fish
  fishy.performMove();
}
```



Now let's assume we want to add class Cricket

We can add a new class called Cricket and a new strategy called Jumpber without making any changes to the other classes.

```
class Jumper implements MoveStrategy {
        public void move(String s) {
            System.out.println("Jumping" + " " + s);
        }
}
class Cricket extends Animal {
     public Cricket(double w) {
          super(w);
          name = "Cricket";
         moveStyle = new Jumper();
     }
}
```

Client using new algorithm for a Cricket Object

```
public class Game
{
    public static void main (String [] s)
   {
        Cricket fast = new Cricket(5);
        // prints- Jumping Cricket
        fast.performMove();
     }
}
```

Now let's assume we want to change the cricket move behavior dynamically: Changing from jumping, to walking

```
public class Game
{
    public static void main (String [] s)
    {
        Cricket fast = new Cricket(5);
        // prints- Jumping Cricket
        fast.performMove();
        fast.setMoveStrategy(new Walker());
        // prints - Walking Cricket:
        fast.performMove();
    }
}
```

Summary of the Lessons Learned

- To be able to change the objects behavior without any changes to the core code of our game (Animal Hierarchy):
  - Separate changeable behaviors
  - Program to interface not implementation
  - Create concrete classes responsible for changeable behaviors.

Benefits and Drawbacks of Strategy Pattern

- Benefits:
  - A hierarchy of Strategy classes creates a family of algorithms that are reusable.
  - It is better than sub-classing, as changeable behaviors are not hardwired into context.
  - Code can be cleaner and sometime reduces that complexity of selecting a desired behavior at the runtime.
  - Can provide different implementation of the same behavior.
- Some drawbacks:
  - Communication overhead.
  - Increased number of classes, and consequently objects in the application.