Design Patterns

Design challenges

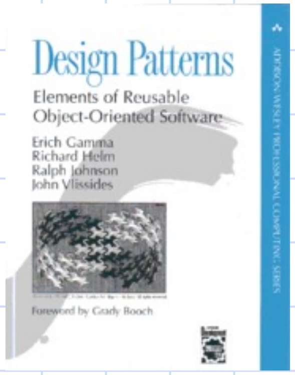
- **Requirement** changes is the number one challenge
- The design **process** is challenging because essential design process is often done in an ad-hoc manner.
- **Technology** changes fast and ever-changing.
- Designers are constantly faced with numerous pressures from stakeholders.
  – Competition 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


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

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

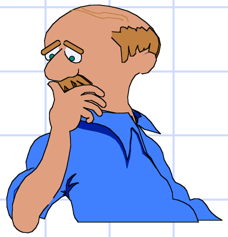
- **Examples of Creational Patterns**
  (abstracting the object-instantiation process)
  - Factory Method
  - Builder
  - Abstract Factory
  - Singleton
  - Prototype
- **Examples Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - Adapter
  - Bridge
  - *Composite*
  - *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: a Walkthrough

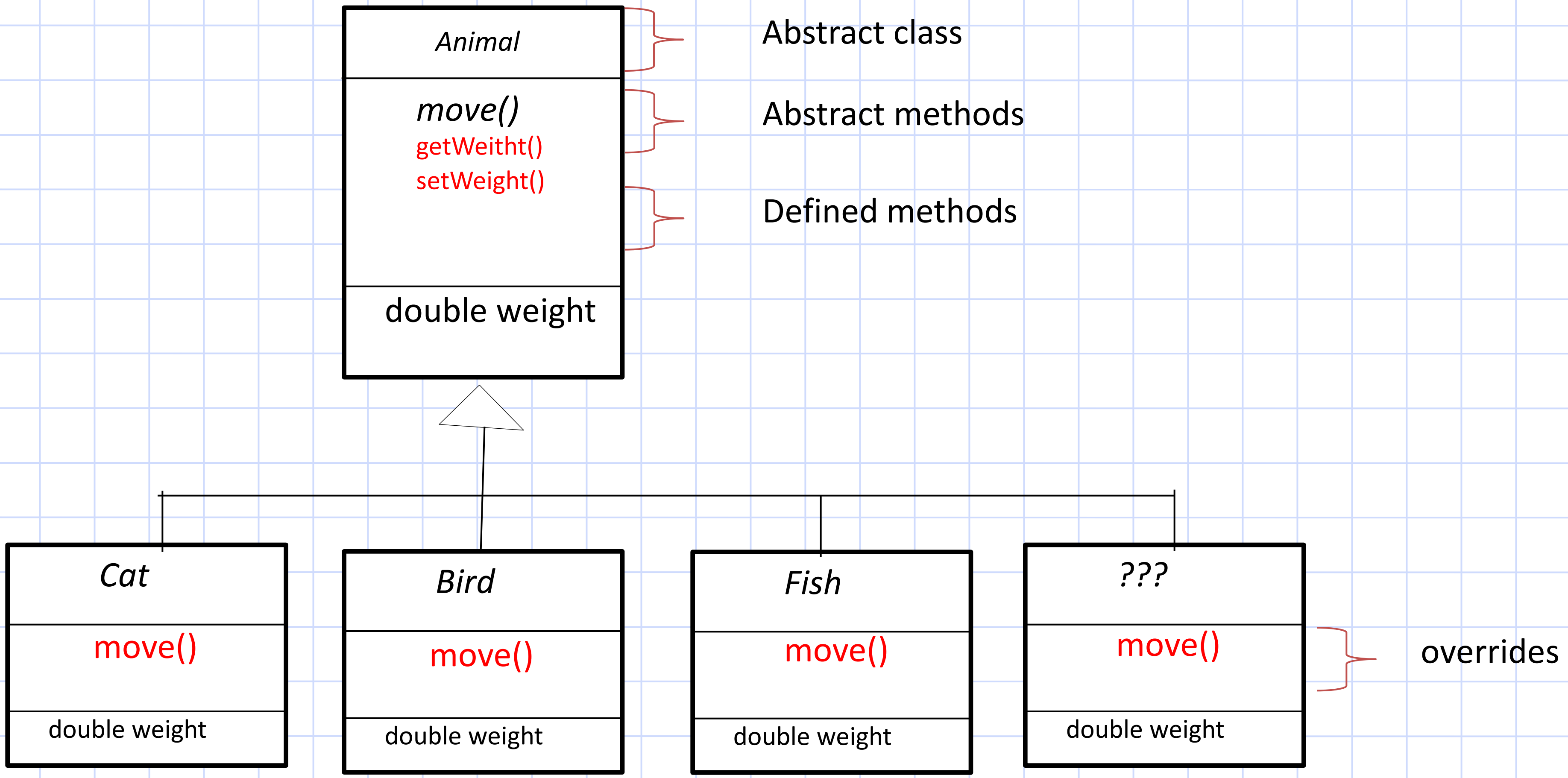
Discussions



- Let's 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 that the "requirements change" is a constant need in software development.**

- Let's 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");
        }
}
```

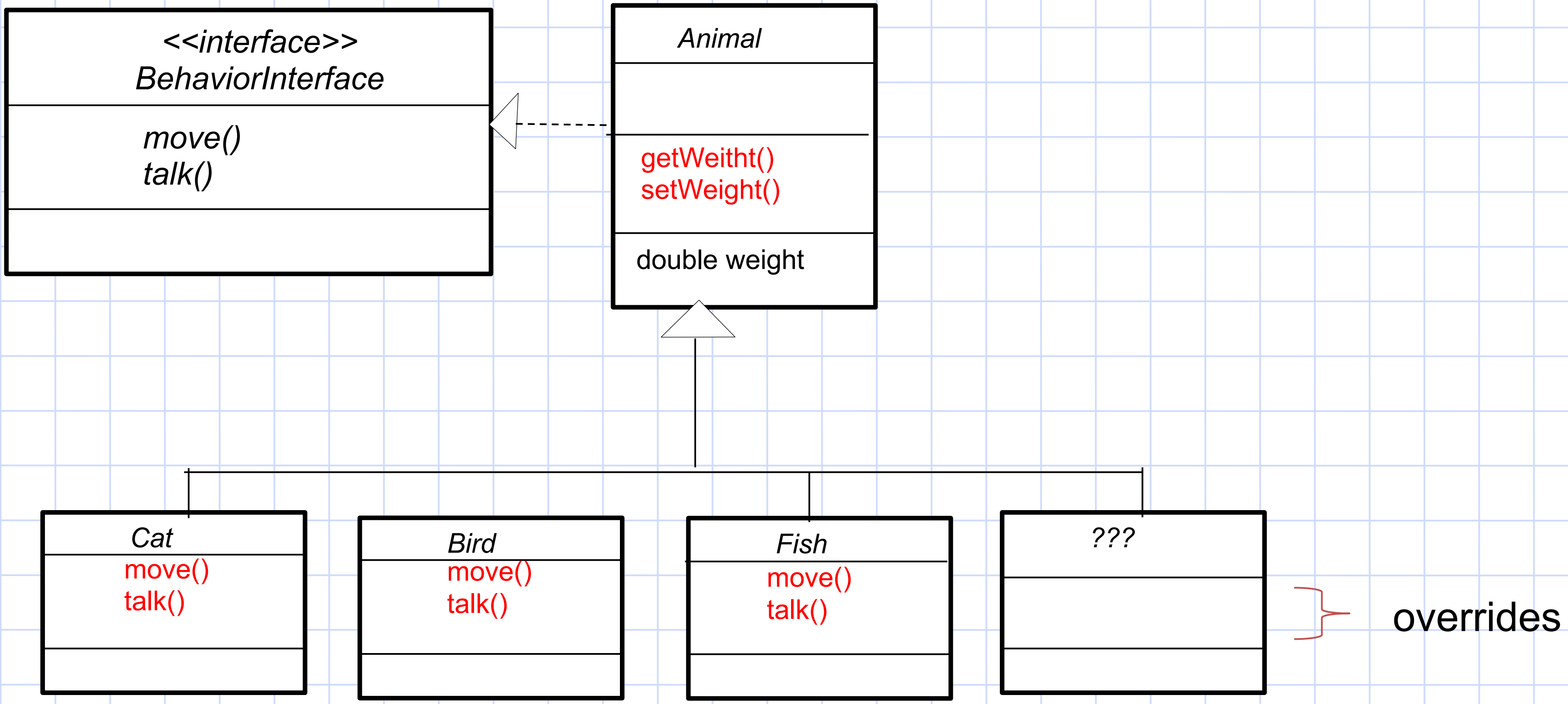
```
class Bird extends Animal {
        public void move() {
                System.out.println(”Flying");
        }
        public void talk() {
                System.out.println(”Tweeting");
        }
}

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


}
```

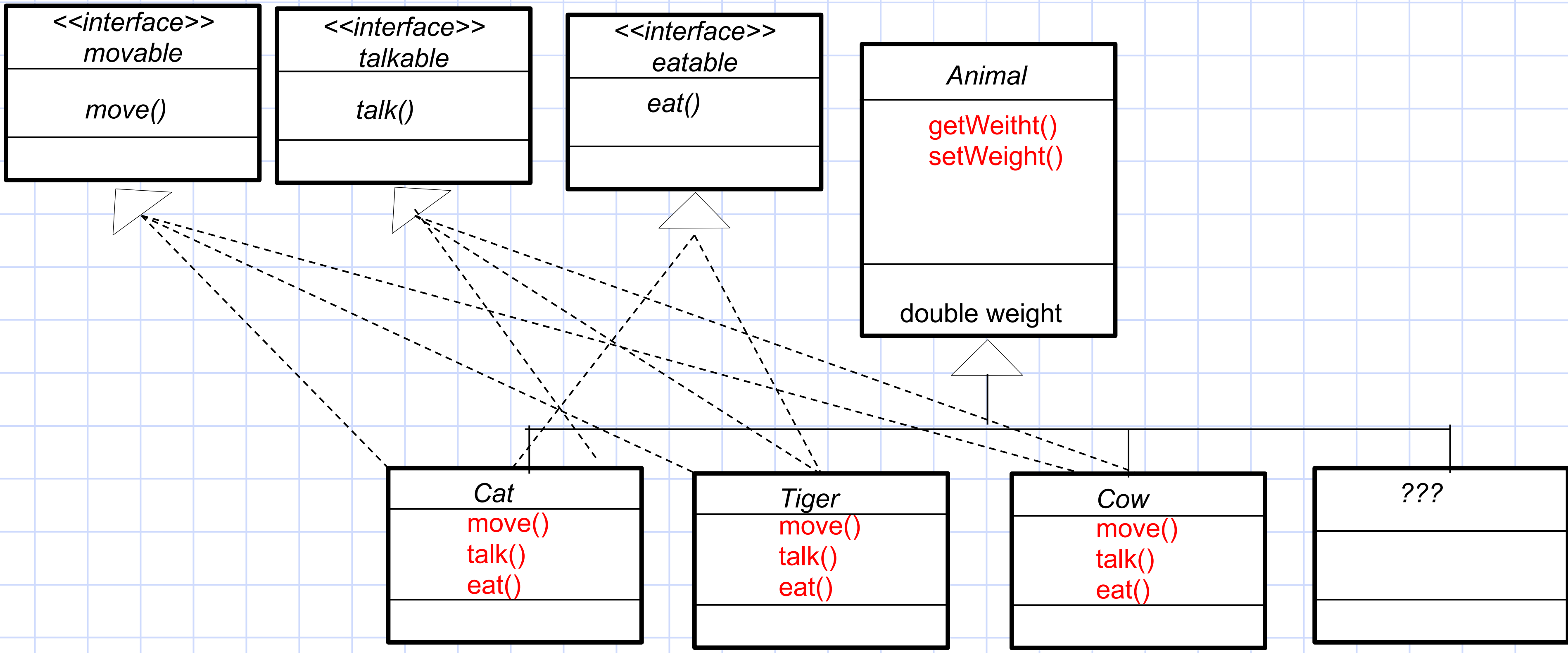
This solution requires too many change

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

- Let's look at other possible issues.
- What if we need to dynamically (at the runtime) change the ability of an animal to do something.
- What if at some point, we want to produce a type of 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.