CODE WRESTLING PRESENTS

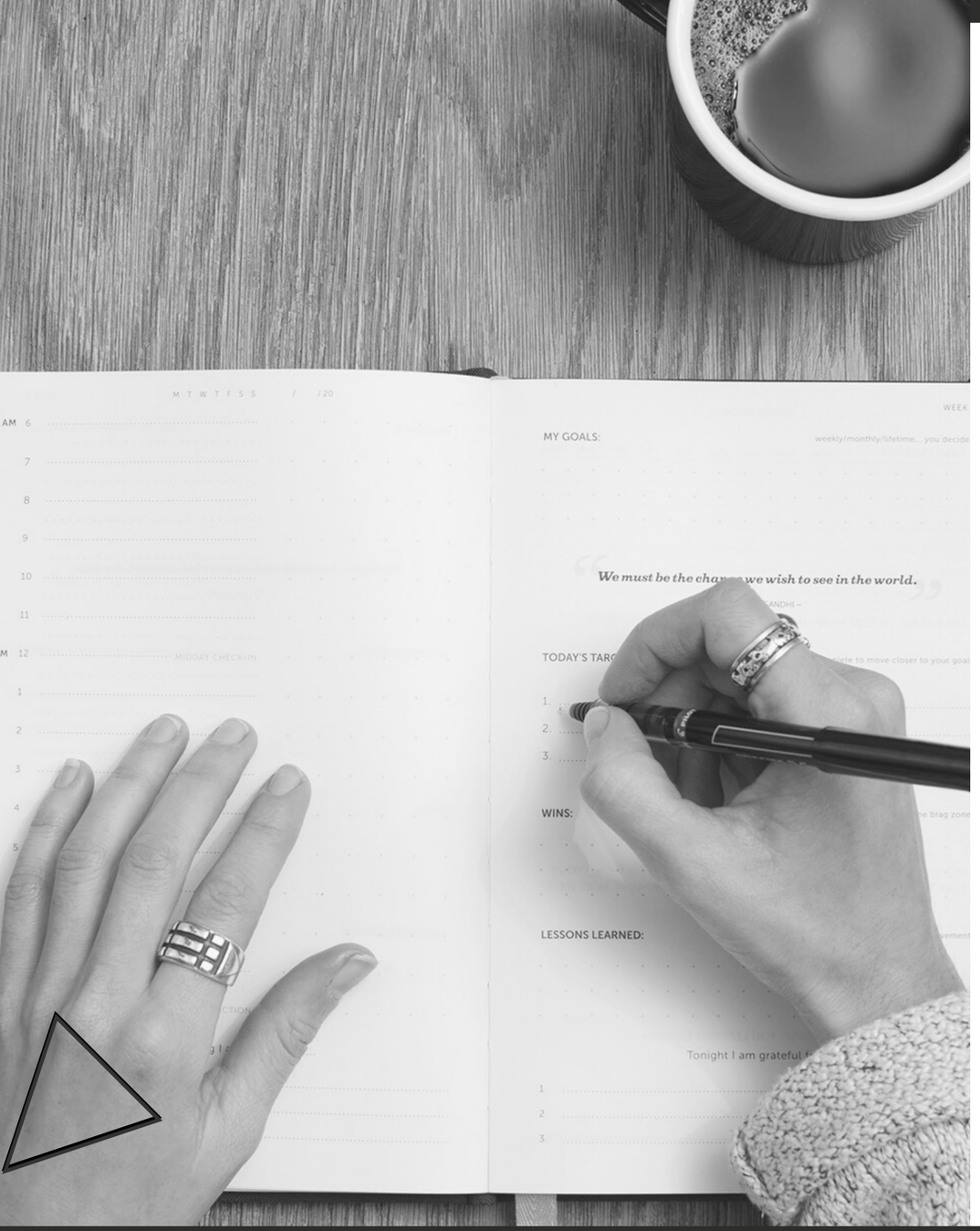
Strategy Design Pattern

RealTime Example of Clash of Clans

Design troops for Clash of Clans

REQUIREMENTS:

1. Two troops: Barbarian and Giant.
2. Both have same value for paramters:
    - TrainingTime.
    - TrainingCost.
    - Hitpoints (Life).
    - FocusOfAttack.
    - AttackingRate.



Before we begin

HOW DO YOU APPROACH PROBLEMS?

1. Two troops: Barbarian and Giant.
2. Both have same same value for paramters:
   - TrainingTime.
   - TrainingCost.
   - Hitpoints (Life).
   - FocusOfAttack.
   - AttackingRate.

Troops

+ Hitpoints(): Integer
+ AttackingRate(): Integer
+ Focus(): void
+ TrainingTime(): TimeSpan
+ Display(): void

Barbarian

+ Display() : void

Giant

+ Display() : void

Let's start coding.

Design troops for Clash of Clans

NEW REQUIREMENTS:

1. Introduce new Troops that can fly.
    - Dragons
    - Balloons

Troops

+ Hitpoints(): Integer
+ AttackingRate(): Integer
+ Focus(): void
+ **Fly() : void**
+ TrainingTime(): TimeSpan
+ Display(): void

Dragon

+ Display() : void

Balloon

+ Display() : void

Troops

+ Hitpoints(): Integer
+ AttackingRate(): Integer
+ Focus(): void
+ **Fly(): void**
+ TrainingTime(): TimeSpan
+ Display(): void

Barbarian

+ Display() : void

Giant

+ Display() : void

Dragon

+ Display() : void

Balloon

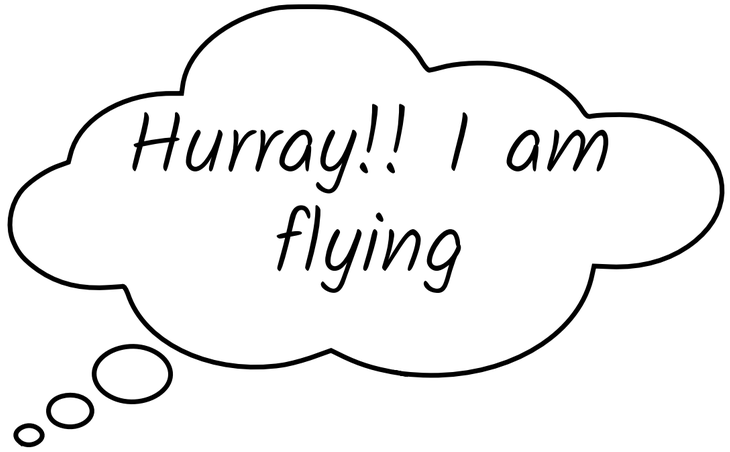
+ Display() : void

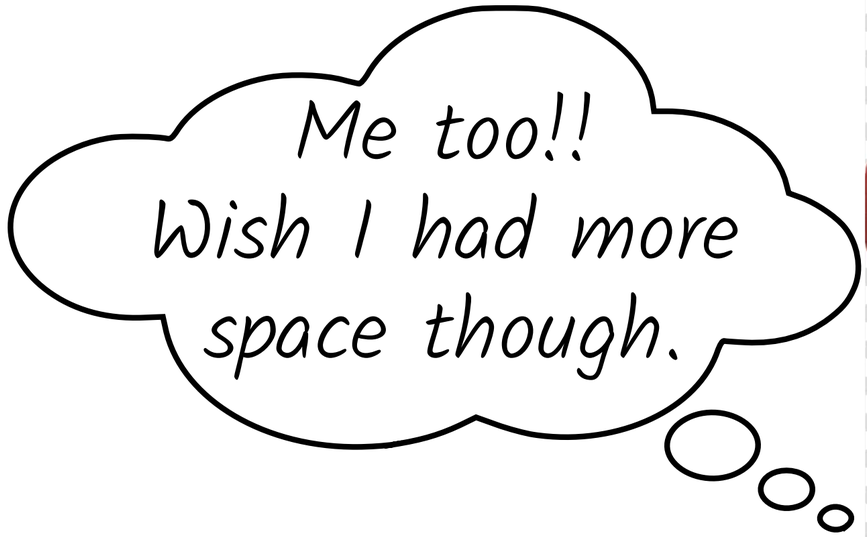
Oops!

Something Went Horribly Wrong!!

What wrong we did?

We gave flying ability to Barbarian and Giant.





Troops

+ Hitpoints(): Integer
+ AttackingRate(): Integer
+ Focus(): void
+ **Fly(): void**
+ TrainingTime(): TimeSpan
+ Display(): void

Barbarian

+ Display() : void

Giant

+ Display() : void

Dragon

+ Display() : void

Balloon

+ Display() : void

By putting fly() in the superclass, you gave flying ability to ALL troops, including those that shouldn't.

What to do now?

I could always just override the fly() method in Barbarian and Giant class

Barbarian

```
void Display() {
    // I look like barbarian
}

void Fly(){
    // I can't fly
}
```

Giant

```
void Display() {
    // I look like Giant
}

void Fly(){
    // I can't fly
}
```

But then what happens when we add new troop that can swim?

Troops

+ Hitpoints(): Integer
+ AttackingRate(): Integer
+ Focus(): void
+ **Fly(): void**
+ **Swim(): void**
+ TrainingTime(): TimeSpan
+ Display(): void

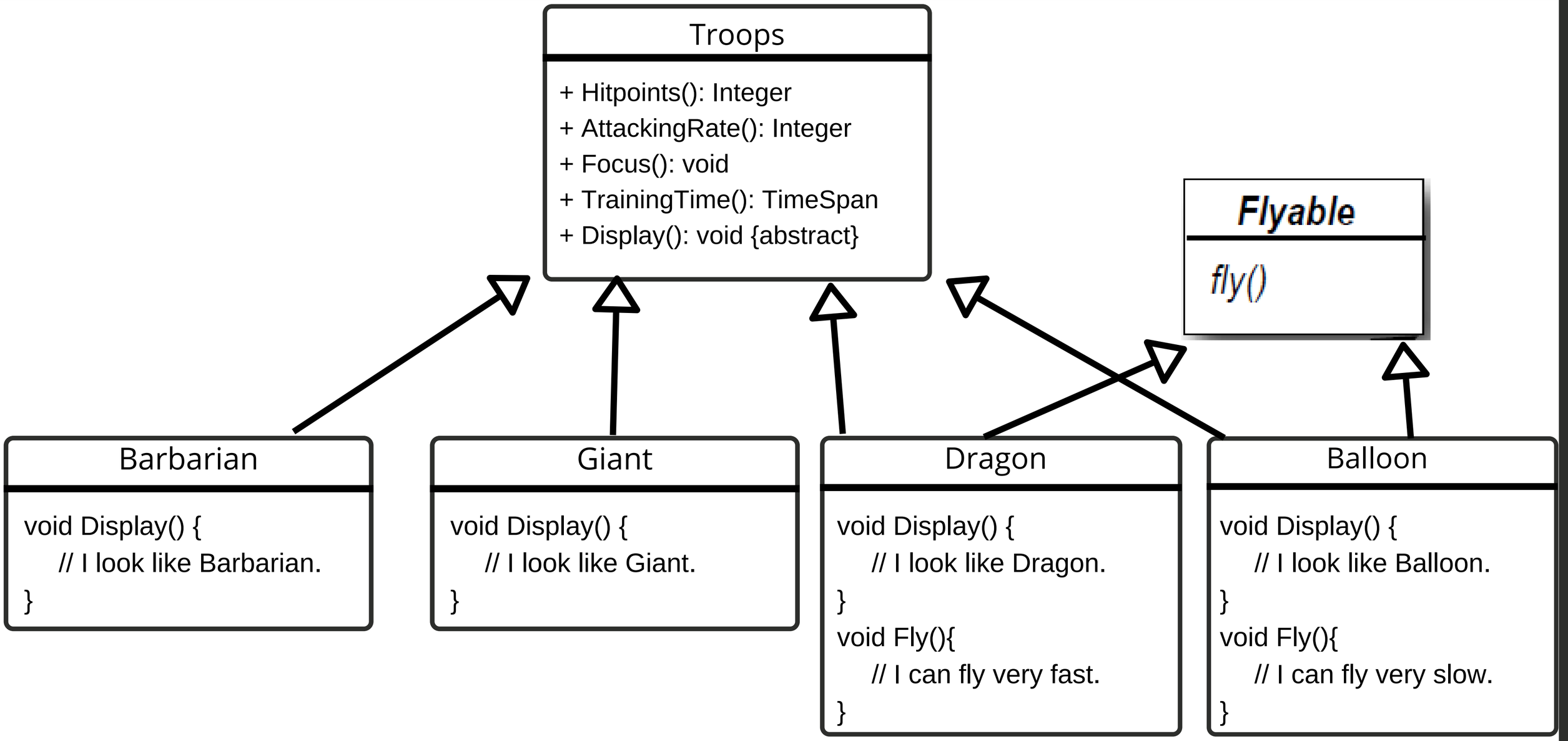
Barbarian

```
void Display() {
    // I look like barbarian
}
void Fly(){
    // I can't fly
}
void swim(){
    // override to do nothing
}
```

Giant

```
void Display() {
    // I look like Giant
}
void Fly(){
    // I can't fly
}
void swim(){
    // override to do nothing
}
```

I could take the **fly()** out of the Troops superclass, and make a **Flyable()** interface with a **fly()** method. That way, only the troops that are supposed to fly will implement that interface and have a **fly()** method.


Troops
+ Hitpoints(): Integer
+ AttackingRate(): Integer
+ Focus(): void
+ TrainingTime(): TimeSpan
+ Display(): void {abstract}
Flyable
fly()
Barbarian
void Display() {
// I look like Barbarian.
}
Giant
void Display() {
// I look like Giant.
}
Dragon
void Display() {
// I look like Dragon.
}
void Fly(){
// I can fly very fast.
}
Balloon
void Display() {
// I look like Balloon.
}
void Fly(){
// I can fly very slow.
}

GUIDELINES TO DESIGN:

1

TAKE WHAT VARIES AND “ENCAPSULATE” IT SO IT WON’T AFFECT THE REST OF YOUR CODE.

- Figure out the areas in your code which are going to change and separate them.
- The result? Fewer unintended consequences from code changes and more flexibility in your systems!
- Here, TypeOfAttack, MovementOfTroop, IncreaseWRTLevels is changing,

GUIDELINES TO DESIGN:

2

ALWAYS PROGRAM TO AN INTERFACE. BUT WHY?

Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();

Cat c = new Cat();
c.meow();
```

Programming to an interface/supertype

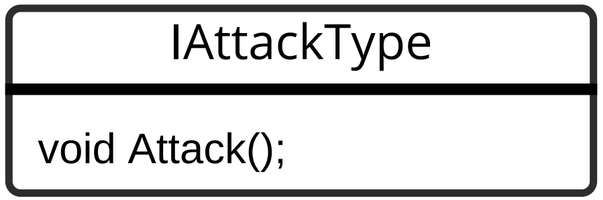
```
Animal animal = new Dog();
animal.makeSound()

animal = new Cat();
animal.makeSound();
```

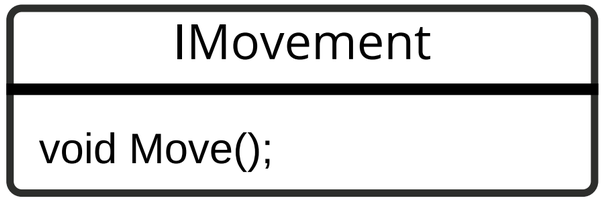
LET'S THINK WITH A BROAD PERSPECTIVE:

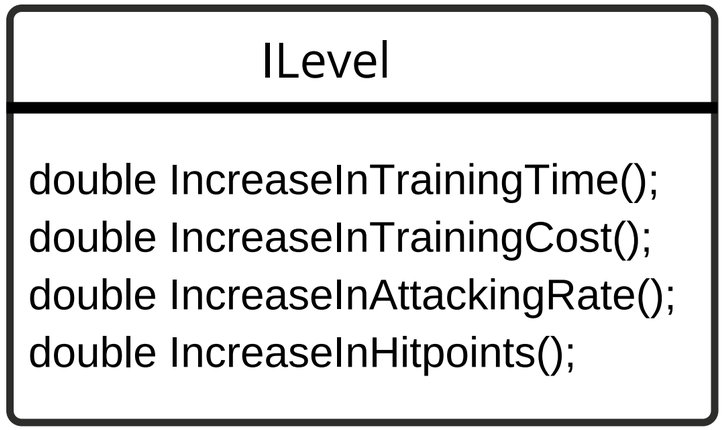
REQUIREMENTS:

1. Each troop has its own type of Attack (Sword Attack, Punch Attack, Flame Attack, Bomb Attack).
2. Each troop has its own intensity of attack() and Hitpoint i.e life.
3. A particular troop can move on ground or can fly.
4. Now you have housing space which is same for all the troops.
5. Upgrade, as level increases, the intensity of attack and life increases.

LET'S DESIGN THE TROOPS

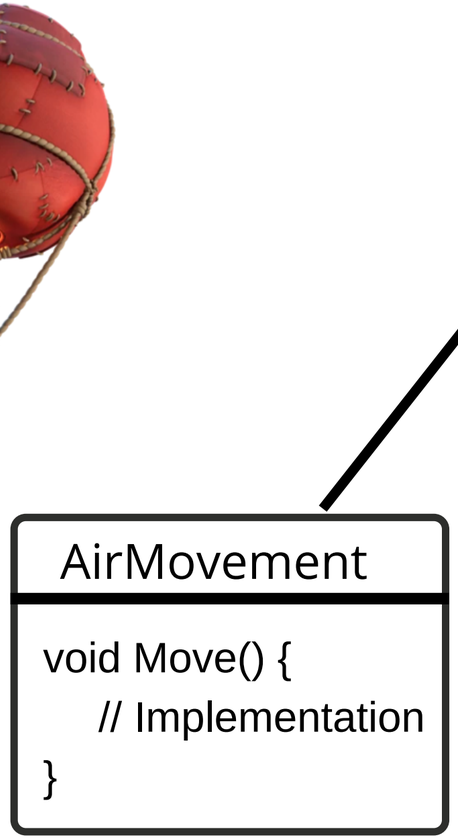






LET'S DESIGN THE TROOPS

LET'S DESIGN THE TROOPS



ILevel

```
double IncreaseInTrainingTime();
double IncreaseInTrainingCost();
double IncreaseInAttackingRate();
double IncreaseInHitpoints();
```

LevelOne

```
double IncreaseInTrainingTime(){
}
double IncreaseInTrainingCost(){
}
double IncreaseInAttackingRate(){
}
double IncreaseInHitpoints(){
}
```

LevelTwo

```
double IncreaseInTrainingTime(){
}
double IncreaseInTrainingCost(){
}
double IncreaseInAttackingRate(){
}
double IncreaseInHitpoints(){
}
```

Troops

IAttackType attackType;
IMovement movement;
ILevel level;

```
public int HousingSpace() {
  return 5;
}
public abstract void Attack();
public abstract double TrainingTime();
public abstract double TrainingCost();
public abstract double Hitpoints();
public abstract void Display();
```

Barbarian

// implement all abstract methods

Giant

// implement all abstract methods

Dragon

// implement all abstract methods

Balloon

// implement all abstract methods

Congratulations on your first pattern!

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.

Congratulations on your first pattern!

Always prefer **Has a** relationship over **Is a** relationship.

Thank you so much

Happy Learning !!!

FOR INQUIRIES AND CONCERNS:

codewrestling@gmail.com

www.codewrestling.com

https://t.me/codewrestling