2. Object-oriented programming

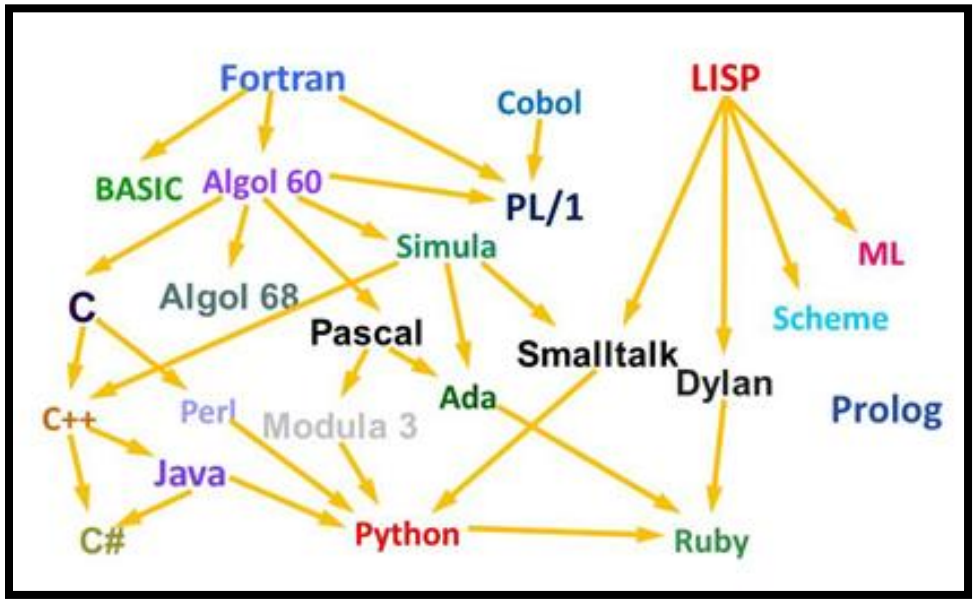
Game Player Experience Design

Prof. H. Kang

Programming Language for Game

Programming languages in the game industry

- Since C++ is arguably the most commonly used language in the game industry, we will focus primarily on C++ in this lecture.
- However, many other languages are also used in the game industry.
  - Imperative languages: C
  - Object-oriented languages: C# and Java
  - Scripting languages: Python, Lua, and Perl
  - Functional languages: Lisp, Scheme, and F#
  - ...



a family tree of language[lan]

Programming Language for Game

Programming languages in the game industry

- It is recommended that every programmer learn at least two high-level languages and an assembly language.
- Low-level languages:
  - Low-level languages generally provide little or no abstraction from a computer’s instruction set (command, function, etc.).
  - Low-level languages require the engineer to handle memory management.
  - C language is considered low-level but true low-level languages are Assembly and Machine code.

Programming Language for Game

Programming languages in the game industry

```
8B542408 83FA0077 06B80000
0000C383 FA027706 B8010000
00C353BB 01000000 B9010000
008D0419 83FA0376 078BD989
C14AEBF1 5BC3
```

```
_fib:
    movl $1, %eax
    xorl %ebx, %ebx
.fib_loop:
    cmpl $1, %edi
    jbe .fib_done
    movl %eax, %ecx
    addl %ebx, %eax
    movl %ecx, %ebx
    subl $1, %edi
    jmp
.fib_loop .fib_done:
    ret
```

A function in hexadecimal representation of
32-bit x86 machine code and x86-64 assembly
code to calculate the *n*th Fibonacci number

Programming Language for Game

Programming languages in the game industry

- High-level languages:
  - High-level languages provide a strong abstraction from the detail of the computer.
  - Generally, they use natural language elements to make coding easier.
  - They also automate (or hide) significant areas of the computing system (e.g. memory management).
  - Therefore, any language in which memory management or garbage collection is done for you is a high-level language.
  - Python, C#, and Java are examples of high-level languages.

Programming Language for Game

Programming languages in the game industry

- High-level languages - Python:
  - Python is one of the most popular programming languages today and is easy to learn and use.
  - It has high flexibility and high readability.
  - Furthermore there are a low of tutorials and sample codes.

Programming Language for Game

Programming languages in the game industry

- High-level languages - Go:
  - Go was developed by Google in 2007.
  - It is syntactically similar to C, but with memory safety, garbage collection, and structural typing[?].

[?] Structural Typing?

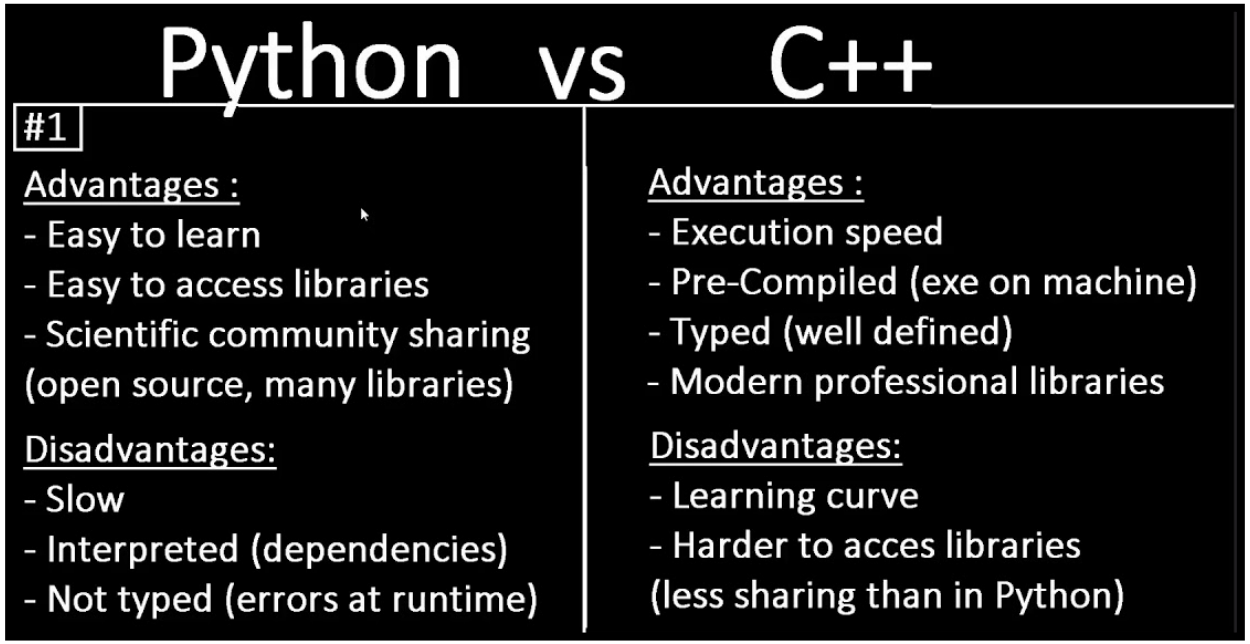
This is a major class of type systems in which type compatibility and equivalence are determined by the type's actual structure or definition. When comparing types, TypeScript only takes into account the members of the type.

```
interface Ironman {
  name: string;
}

class Avengers {
  name: string;
}

let i: Ironman;
i = new Avengers(); // OK, because of structural typing
```

C++ Review

The advantage of C++ over other languages are as follows[C++]:

- It is deterministic (you can determine what gets executed when). This is not true for garbage collected languages and it is not (or partially) true for languages run in virtual machines.
- It offers high performance (you can get C-like performance and even drop in some assembly code if you feel like it).
- It offers enough abstraction to be higher level than other fast languages (like C for example).

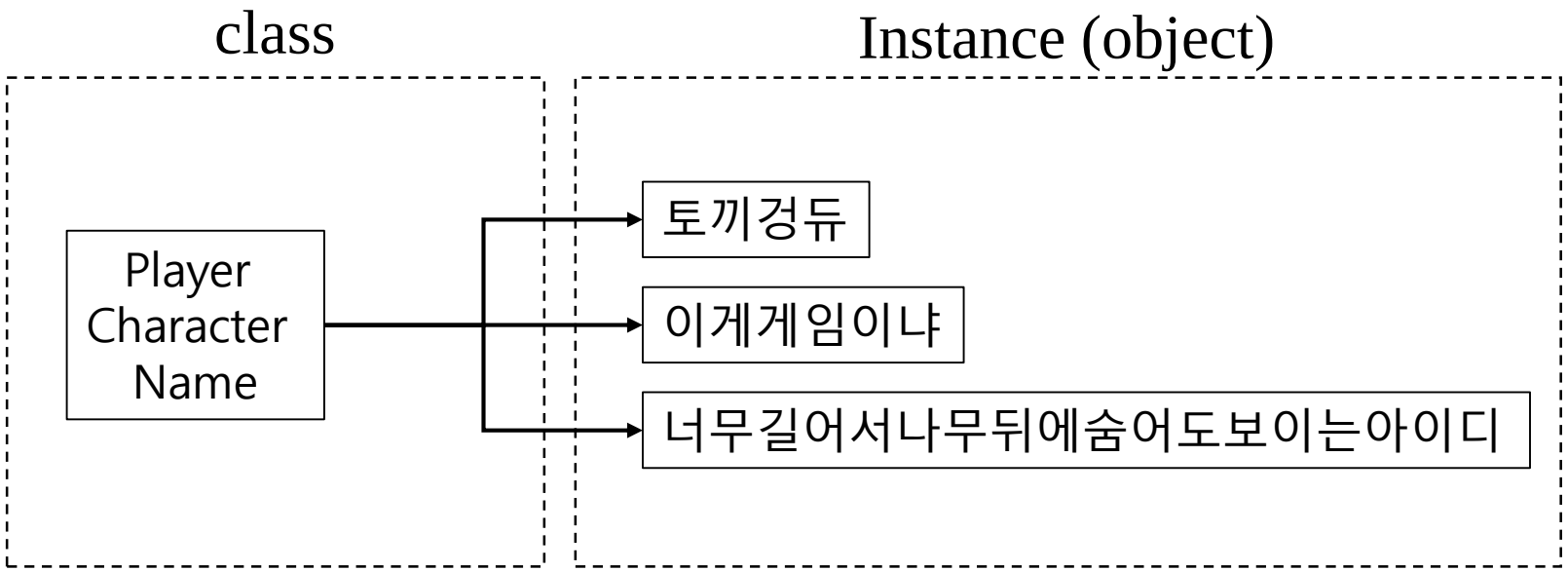


Python vs. C++[comp]

OOP

Classes and objects

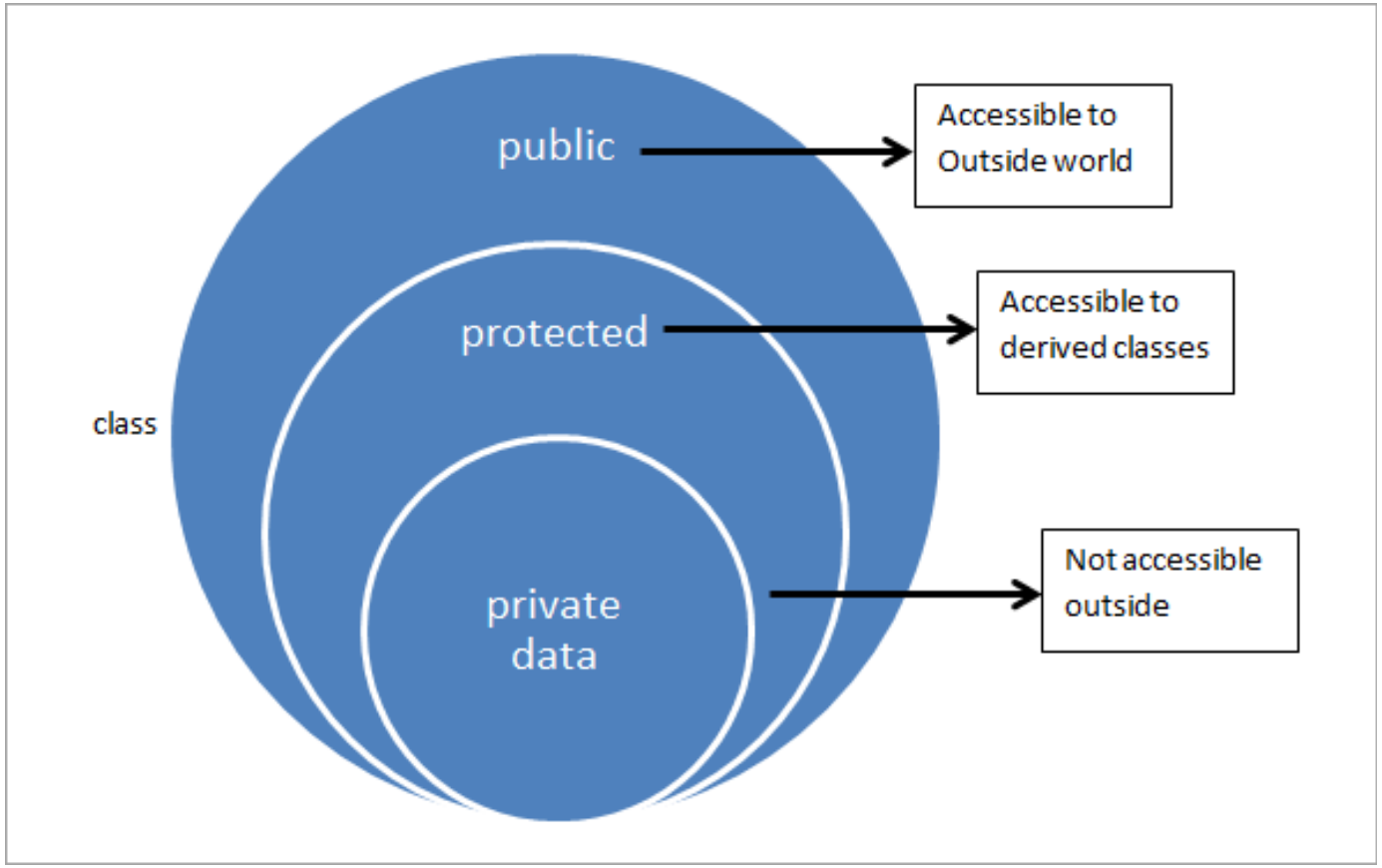
- A class is a collection of attributes (data) and behaviors (code) that together form a useful, meaningful whole.
- A class is a specification describing how individual instances of the class, known as objects, should be constructed.
- For example, your pet ‘토끼겅듀’ is an instance of the class ‘player character’.



OOP

Encapsulation

- Encapsulation means that an object presents only a limited interface to the outside world; the object's internal state and implementation details are kept hidden.
- Encapsulation simplifies life of programmers because only the limited interfaces of the class need to be understood, not potentially complex implementation details.

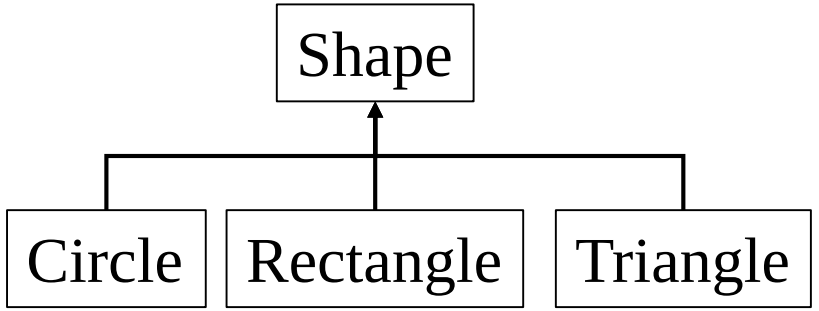


encapsulation in C++[encap]

OOP

Inheritance

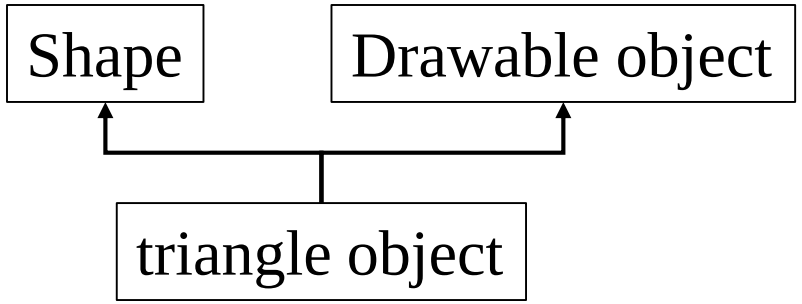
- Inheritance allows new classes to be defined as extensions to preexisting classes.
- The new class modifies or extends the data, interface and/or behavior of the existing class.
- If class *child* extends class *parent*, we say the *child* inherits from or is derived from *parent*. In this relationship, the class *parent* is known as the base class or superclass, and the class *child* is the derived class or subclass.
- Inheritance creates an “is-a” relationship between classes.
  - A circle is a type of shape.
  - A rectangle is a type of shape.
  - A triangle is a type of shape.



OOP

Multiple Inheritance

- Some languages support multiple inheritance (MI), meaning that a class can have more than one parent class.
- In theory MI can be quite elegant, but in practice this kind of design usually gives rise to a lot of confusion and technical difficulties.
- MI transforms a simple *tree* of classes into a potentially complex *graph*.

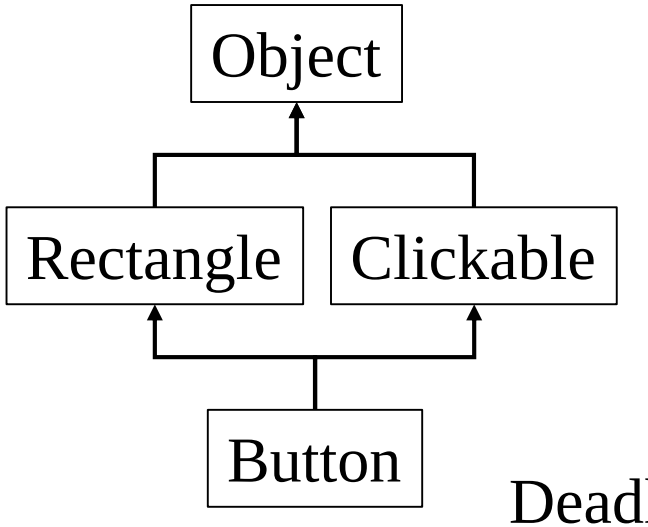


multiple inheritance

OOP

Multiple Inheritance

- The “diamond problem” is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
  - In the context of GUI, a class *Button* may inherit from both classes *Rectangle* (for appearance) and *Clickable* (for functionality).
  - Classes *Rectangle* and *Clickable* both inherit from the *Object* class.
  - If the *equals* method is called for a *Button object* and there is no such method in the *Button* class but there is an overridden equals method in *Rectangle* or *Clickable*, which method should be eventually called?

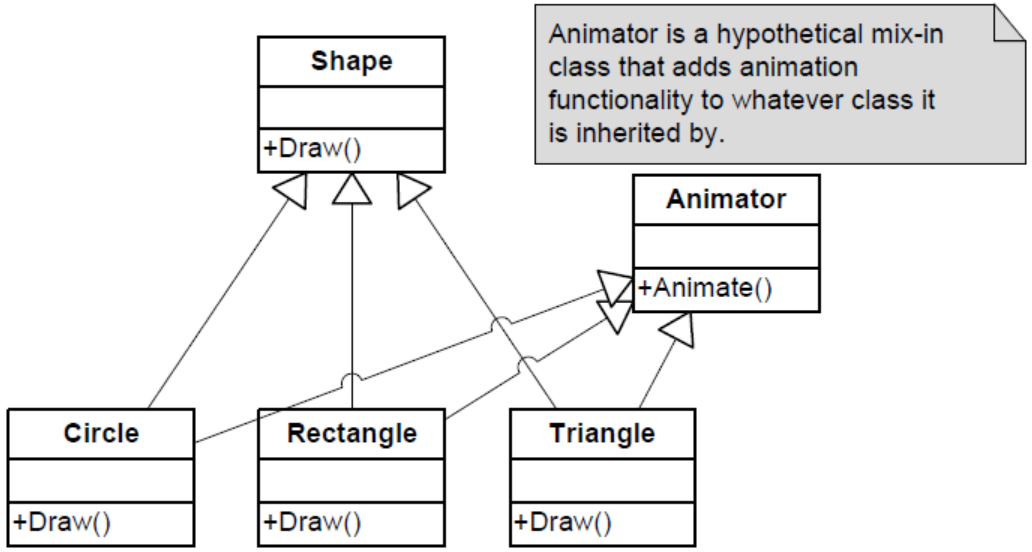


Deadly Diamond of Death[DDD]

OOP

Multiple Inheritance

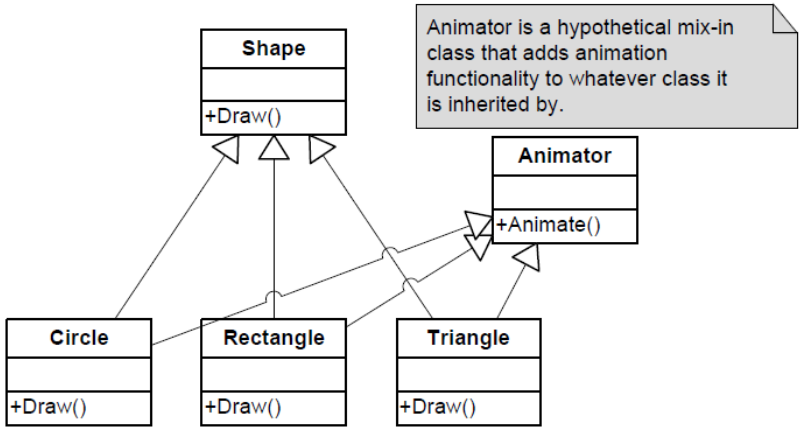
- Most C++ software developers avoid multiple inheritance completely or only permit it in a limited form.
- A *Mixin* is a special kind of multiple inheritance.
  - A mixin class acts as the parent class, containing the desired functionality.
  - A subclass can then inherit or simply reuse this functionality.
  - Typically, the mixin will export the desired functionality to a child class, without creating a rigid, single “is a” relationship.



OOP

Multiple Inheritance

- A *Mixin* provides several advantages:
  - It provides a mechanism for multiple inheritance by allowing one class to use common functionality from multiple classes, but without the complex semantics of multiple inheritance.
  - It encourage code reuse. Instead of repeating the same code over and over again, the common functionality can simply be grouped into a mixin and then included into each class that requires it.
  - Mixins allow inheritance and use of only the desired features from the parent class, not necessarily all of the features from the parent class.



OOP

Polymorphism

- Polymorphism is a language feature that allows a collection of objects of different types to be manipulated through a single common interface.
- For example, a 2D painting program might be given a list of various shapes to draw on-screen.

- One way to draw this heterogeneous collection of shapes is to use a switch statement to perform different drawing commands for each distinct type of shape.

```
void drawShapes(std::list<Shape*> shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      switch (pShape->mType)
      {
      case CIRCLE:
         // draw shape as a circle
         break;

      case RECTANGLE:
         // draw shape as a rectangle
         break;

      case TRIANGLE:
         // draw shape as a triangle
         break;

      //...
      }
   }
}
```

OOP

Polymorphism

- The problem with this approach is that the drawShapes() function needs to know about all of the kinds of shapes that can be drawn.
- This is fine in a simple example, but as our code grows in size and complexity, it can become difficult to add new types of shapes to the system.
- Whenever a new shape type is added, one must find every place in the code base where knowledge of the set of shape types is embedded and add a case to handle the new type.

```
void drawShapes(std::list<Shape*> shapes)
{
    std::list<Shape*>::iterator pShape = shapes.begin();
    std::list<Shape*>::iterator pEnd = shapes.end();

    for ( ; pShape != pEnd; pShape++)
    {
        switch (pShape->mType)
        {
        case CIRCLE:
            // draw shape as a circle
            break;

        case RECTANGLE:
            // draw shape as a rectangle
            break;

        case TRIANGLE:
            // draw shape as a triangle
            break;

        //...
        }
    }
}
```

OOP

Polymorphism

- The solution is to insulate the majority of our code from any knowledge of the types of objects with which it might be dealing.
- To accomplish this, we can define classes for each of the types of shapes we wish to support.
- All of these classes would inherit from the common base class *Shape*.
- A virtual function would be defined called Draw(), and each distinct shape class would implement this function in a different way.
- Without knowing what specific types of shapes it has been given, the drawing function can now simply call each shape's Draw() function in turn.

```
struct Shape
{
   virtual void Draw() = 0; // pure virtual function
};

struct Circle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a circle
   }
};

struct Rectangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a rectangle
   }
};

struct Triangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a triangle
   }
};
void drawShapes(std::list<Shape*> shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      pShape->Draw(); // call virtual function
   }
}
```

OOP

Composition and Aggregation

- Composition is the process of creating complex one from simpler ones.
  - Composition creates a “owns-a” relationship between classes.
  - “*I own an object and I am responsible for its lifetime.*”
- Aggregation is a process in which one class defines another class as any entity reference.
  - Aggregation creates a “has-a” relationship between classes.
  - “*I have an object which I’ve borrowed from someone else. When I die, the object may live on.*”

OOP

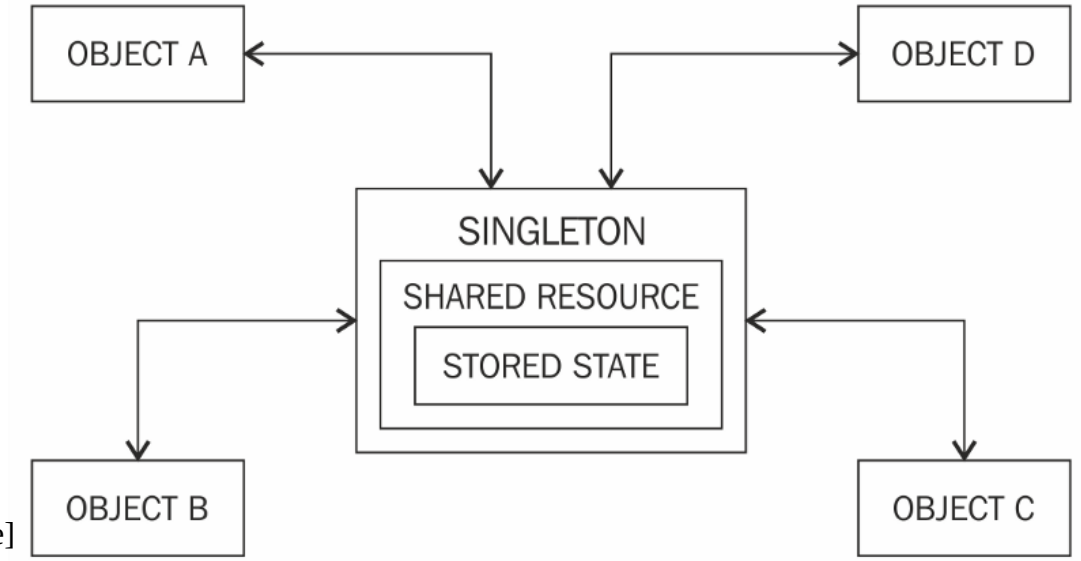
Composition and Aggregation

- Example:
  - public class Window {//composition
    private Rectangle rect = new Rectangle();
    }
  - public class Window {//aggregation
    private Rectangle rect;
    void setRect(Rectangle rectParam) {
    this.rect = rectParam;
    }
    }

OOP

Design Patterns

- When the same type of problem arises over and over, and many different programmers employ a very similar solution to that problem, we say that a *design pattern* has arisen.
- In object-oriented programming, a number of common design patterns have been identified and described by various authors.
  - **Singleton**: This pattern ensures that a particular class has only one instance and provides a global point of access to it. A singletone usually encapsulates a unique resources and makes it readily available throughout the application. The resource might be hardware, a network service, a persistent store, or anything else that can be modeled as a unique object or service.

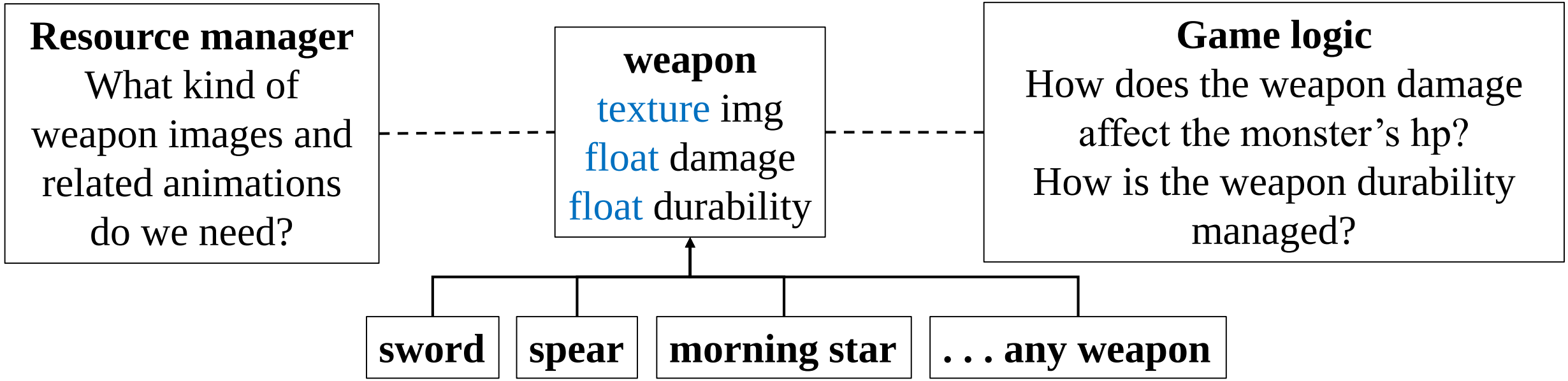


singleton pattern[single]

OOP

OOP Example in the Field

- Assume that you already had a ‘weapon’ class.
  - Damage, Durability, etc.
- Assume that you also implemented the game logic to reduce the monster’s health point based on the user’s ‘weapon’ damage.
- Then, other programmers can define various types of weapons such as spear, sword, or morning star without considering the detail of the weapon class and game logic.
  - They just fill in the variables (damage, durability, etc.)



Reference

- [lan] https://medium.com/@christianreyompad/other-variations-of-c-3a78634e7891
- [C++] https://stackoverflow.com/questions/3318898/why-is-c-that-powerful-concerning-game-development#:~:text=The%20main%20reason%20it's%20used,game%20development%20are%20C%2B%2B%20APIs.&text=Another%20trend%20worth%20noting%20is,much%20of%20the%20game%20logic.
- [comp] https://www.youtube.com/watch?v=jeg1haA3Eis&ab_channel=DataScientist
- [encap] https://www.softwaretestinghelp.com/encapsulation-in-cpp/
- [DDD] Martin, Robert C. (1997-03-09). "Java and C++: A critical comparison" (PDF). Objectmentor.com. Archived from the original (PDF) on 2005-10-24. Retrieved 2016-10-21.
- [itor] https://en.wikipedia.org/wiki/Iterator_pattern
- [abs] https://www.bogotobogo.com/DesignPatterns/abstractfactorymethod.php