1. 虚拟类继承方式的代码修改量少，通用性高，更适用于软件复用。相对而言，普通方式的代码修改量大，通用性低，在大规模软件复用中效果不佳。

2.

假设有三个特性（模块）， ModuleA, ModuleB, ModuleC。有两个类 ClassA 和 ClassB 需要使用它们。ClassA 会用到 ModuleA 和 ModuleB，ClassB 会用到全部三个特性。而且我们还假设现在全部A，B和C模块会只被ClassA和ClassB用到。

继承：

class ClassA {

functions of ModuleA;

functions of ModuleB;

};

class ClassB extends ClassA {

functions of ModuleC;

};

组合：

class ModuleA {};

class ModuleB {};

class ModuleC {};

class ClassA {

ModuleA ma;

ModuleB mb;

};

class ClassB {

ModuleA ma;

ModuleB mb;

ModuleC mc;

};

//或者：

class ClassB {

ClassA ca;

ModuleC mc;

};

两种方法都重用了模块。

对于组合，我们一般把模块叫做组件。在游戏开发中，这样的组合又叫做基于组件的实体系统。

组合比继承有哪些优势？

组合允许将问题分割成互不依赖的子元件。每个元件可以由不同的开发者开发。我们可以让三个程序员来同时开发 ModuleA， ModuleB 以及 ModuleC。

元件可以在运行时被替换，删除，或者添加（动态绑定）。与之相反，继承不能或者非常难（取决于编程语言）在运行时替换，删除，或者添加特性（静态绑定）。

组合比继承有更少的耦合。继承强制了 ClassB 和 ClassA 之间的耦合，但组合不会。

组件可以被用在其它的组合当中用以构成不同的行为。在继承中则没有办法只重用一个单个特性。继承强制了父类的所有特性被带入到了子类。

更少的代码冗余。继承比较容易导致冗余。B 继承自 A，通常 B 只需要 A 的部分特性，不需要的部分就成了冗余。如果 B 是由 A 的部分元件组合来的，那么 B 可以舍弃不需要的元件来避免冗余。

更好的封装。组合是基于公开接口的。每个元件不知道其它元件的内部细节。这正是封装所指的。而继承则把父类的内部细节（保护的接口）透露给了子类。更糟的是，子类通常会对父类做某些假设，而且父类也会假设子类会尊重它要求的接口。

容易更改。在组合中，任何元件都可以被更改而保证对其它元件影响很小，只要保持公开的接口。继承则强制了很紧密耦合的层次链。对于层次链任何节点的改变都将影响整个链。

小模块和原子模块。组合中，大的模块是由小的模块组成的，因此模块的层次可以得到控制。有可能我们只有两个层次，原子模块，和真实的功能模块，因 而层次非常平坦。在继承中，大的模块来自从小的模块继承。想要一个模块？继承吧，得到一个新的层级。这就可能造成有很多的层级，导致系统的模块层次非常复 杂且难以理解。

3.

#include <iostream>

#include <string>

#include <vector>

class Book {

private:

std::string title;

std::string author;

public:

Book(const std::string& title, const std::string& author)

: title(title), author(author) {}

std::string getTitle() const {

return title;

}

std::string getAuthor() const {

return author;

}

};

class Library {

private:

std::vector<Book> books;

public:

void addBook(const Book& book) {

books.push\_back(book);

}

void displayBooks() const {

for (const auto& book : books) {

std::cout << "Title: " << book.getTitle() << ", Author: " << book.getAuthor() << std::endl;

}

}

};

class User {

private:

std::string name;

public:

User(const std::string& name)

: name(name) {}

void borrowBook(Library& library, const std::string& title) {

// 用户只需要与Library类进行交互

for (auto& book : library.books) {

if (book.getTitle() == title) {

std::cout << "User " << name << " borrowed the book: " << book.getTitle() << std::endl;

// 借阅书籍的逻辑

return;

}

}

std::cout << "Book with title " << title << " not found." << std::endl;

}

};

int main() {

Book book1("The Great Gatsby", "F. Scott Fitzgerald");

Book book2("To Kill a Mockingbird", "Harper Lee");

Library library;

library.addBook(book1);

library.addBook(book2);

User user("John");

user.borrowBook(library, "The Great Gatsby");

return 0;

}

一、定义

迪米特法则的定义是：只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。

其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

迪米特法则中的“朋友”是指：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。

二、实现方法

从迪米特法则的定义和特点可知，它强调以下两点：

从依赖者的角度来说，只依赖应该依赖的对象。

从被依赖者的角度说，只暴露应该暴露的方法。

所以，在运用迪米特法则时要注意以下 6 点

在类的划分上，应该创建弱耦合的类。类与类之间的耦合越弱，就越有利于实现可复用的目标。

在类的结构设计上，尽量降低类成员的访问权限。

在类的设计上，优先考虑将一个类设置成不变类。

在对其他类的引用上，将引用其他对象的次数降到最低。

不暴露类的属性成员，而应该提供相应的访问器（set 和 get 方法）。

谨慎使用序列化（Serializable）功能。