Chapter 21

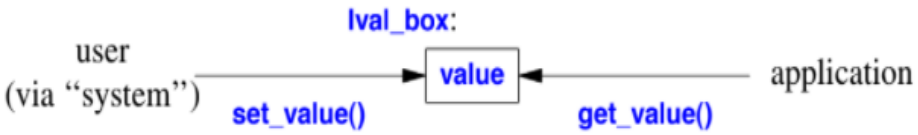
21.1 Introduction

Trọng tâm chính của chương này là các kỹ thuật thiết kế, hơn là các tính năng ngôn ngữ. Các ví dụ được lấy từ thiết kế giao diện người dùng, nhưng tôi tránh chủ đề về lập trình theo hướng sự kiện thường được sử dụng cho các hệ thống giao diện người dùng đồ họa (GUI). Một cuộc thảo luận về cách chính xác một hành động trên màn hình được chuyển thành một lệnh gọi của một hàm thành viên sẽ bổ sung ít vào các vấn đề của lớp thiết kế phân cấp và có khả năng gây mất tập trung rất lớn: nó là một chủ đề thú vị và quan trọng theo đúng nghĩa của nó. Để hiểu về GUI, hãy xem một trong nhiều thư viện C ++ GUI.

21.2 Design of Class Hierarchies

Hãy xem xét một vấn đề thiết kế đơn giản: Cung cấp một cách để một chương trình (‘‘ một ứng dụng ’’) nhận một giá trị nguyên từ người dùng. Điều này có thể được thực hiện theo nhiều cách khác nhau. Để cách ly chương trình của chúng tôi khỏi sự đa dạng này và cũng để có cơ hội khám phá các lựa chọn thiết kế khả thi, chúng ta hãy bắt đầu bằng cách xác định mô hình chương trình của chúng tôi về thao tác nhập đơn giản này.

Ý tưởng là có một lớp Ival_box (‘‘ nhập giá trị số nguyên ’’) biết nó sẽ chấp nhận phạm vi giá trị đầu vào nào. Một chương trình có thể hỏi Ival_box về giá trị của nó và yêu cầu nó nhắc người dùng nếu cần. Ngoài ra, một chương trình có thể hỏi Ival_box nếu người dùng đã thay đổi giá trị kể từ lần cuối chương trình xem xét nó:



Bởi vì có nhiều cách để triển khai ý tưởng cơ bản này, chúng ta phải giả định rằng sẽ có nhiều loại Ival_box khác nhau, chẳng hạn như sliders, plain boxes trong đó người dùng có thể nhập số, quay số và tương tác bằng giọng nói.

Cách tiếp cận chung là xây dựng một " hệ thống giao diện người dùng ảo " để ứng dụng sử dụng. Hệ thống này cung cấp một số dịch vụ được cung cấp bởi các hệ thống giao diện người dùng hiện có. Nó có thể được thực hiện trên nhiều hệ thống khác nhau để đảm bảo tính di động của mã ứng dụng. Đương nhiên, có những cách khác để cách ly một ứng dụng khỏi hệ thống giao diện người dùng. Tôi chọn cách tiếp cận này vì nó chung chung, vì nó cho phép tôi chứng minh nhiều kỹ thuật và sự cân bằng trong thiết kế, bởi vì những kỹ thuật đó cũng là những kỹ thuật được sử dụng để xây dựng các hệ thống giao diện người dùng " thực " và quan trọng nhất - bởi vì những kỹ thuật có thể áp dụng cho các vấn đề vượt xa phạm vi hẹp của hệ thống giao diện.

Ngoài việc bỏ qua chủ đề về cách ánh xạ các hành động (sự kiện) của người dùng với các lệnh gọi thư viện, tôi cũng bỏ qua sự cần thiết của việc khóa trong một hệ thống GUI đa luồng.

21.2.1 Implementation Inheritance

Giải pháp đầu tiên của chúng tôi là một cấu trúc phân cấp lớp sử dụng kế thừa triển khai (như thường thấy trong các chương trình cũ hơn).

Lớp Ival_box xác định giao diện cơ bản cho tất cả các Ival_box và chỉ định triển khai mặc định mà các loại Ival_box cụ thể hơn có thể ghi đè bằng các phiên bản của riêng chúng. Ngoài ra, chúng tôi khai báo dữ liệu cần thiết để triển khai khái niệm cơ bản:

```
class Ival_box {
protected:
int val;
int low, high;
bool chang ed {false}; // được thay đổi bởi người dùng bằng set_value ()
public:
Ival_box(int ll, int hh) :val{ll}, low{ll}, high{hh} { }
virtual int get_value() { chang ed = false; return val; } // cho ứng dụng
virtual void set_value(int i) { chang ed = true; val = i; } //cho người dùng
virtual void reset_value(int i) { chang ed = false; val = i; } // cho ứng dụng
virtual void prompt() { }
virtual bool was_chang ed() const { return chang ed; }
virtual ˜Ival_box() {};
};
```

Việc triển khai mặc định của các chức năng là khá cẩu thả và được cung cấp ở đây chủ yếu để minh họa ngữ nghĩa dự định. Ví dụ, một lớp thực tế sẽ cung cấp một số kiểm tra phạm vi.

Một lập trình viên có thể sử dụng " các lớp ival " như thế này:

```
void interact(Ival_box∗ pb)
{
pb−>prompt(); // cảnh báo người dùng
int i = pb−>get_value();
```

```
if (pb−>was_chang ed()) {
// ... giá trị mới; làm việc gì đó ...
}
else {
// ... làm việc gì khác ...
}
}
void some_fct()
{
unique_ptr<Ival_box> p1 {new Ival_slider{0,5}}; // Ival_slider bắt nguồn từ Ival_box
interact(p1.get());
unique_ptr<Ival_box> p2 {new Ival_dial{1,12}};
interact(p2.get());
}
```

Hầu hết mã ứng dụng được viết dưới dạng (con trỏ tới) Ival_boxes thuần túy như cách tương tác (). Bằng cách đó, ứng dụng không muốn biết về số lượng lớn các biến thể tiềm ẩn của khái niệm Ival_box. Kiến thức của các lớp chuyên biệt như vậy bị cô lập trong tương đối ít chức năng tạo ra các đối tượng như vậy. Điều này cách ly người dùng khỏi những thay đổi trong việc triển khai các lớp dẫn xuất. Hầu hết các mã có thể bị lãng quên bởi thực tế là có nhiều loại Ival_box khác nhau.

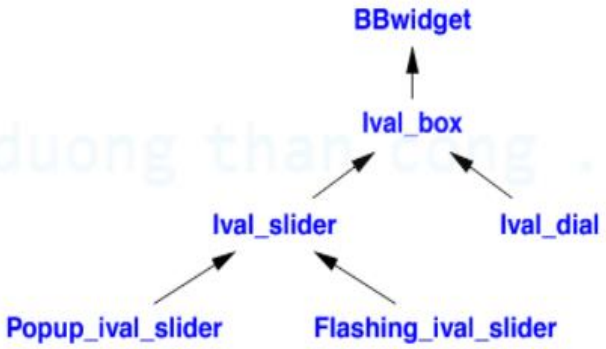
Các loại Ival_box khác nhau được định nghĩa là các lớp bắt nguồn từ Ival_box. Ví dụ:

```
class Ival_slider : public Ival_box {
private:
// ... nội dung đồ họa để xác định “slider” trông như thế nào, v.v. ...
public:
Ival_slider(int, int);
int get_value() override; // lấy giá trị từ người dùng và gửi nó vào val
void prompt() override;
};
```

Hầu hết các hệ thống giao diện người dùng cung cấp một lớp xác định các thuộc tính cơ bản của một thực thể trên màn hình. Vì vậy, nếu chúng ta sử dụng hệ thống từ ‘‘ Big Bucks Inc., chúng tôi sẽ phải làm cho mỗi lớp Ival_slider, Ival_dial, v.v., của chúng tôi trở thành một loại BBwid _get. Điều này đơn giản nhất sẽ đạt được bằng cách viết lại Ival_box của chúng tôi để nó bắt nguồn từ BBwidget.Theo cách đó, tất cả các lớp của chúng ta kế thừa tất cả các thuộc tính của một BBwidget. Ví dụ: mọi Ival_box đều có thể được đặt trên màn hình, tuân theo các quy tắc về kiểu đồ họa, được thay đổi kích thước, được kéo xung quanh, v.v., theo tiêu chuẩn do hệ thống BBwidget đặt ra. Hệ thống phân cấp lớp của chúng ta sẽ trông như thế này:

```
class Ival_box : public BBwidget { /* ... */ }; // được viết lại để sử dụng BBwidget

class Ival_slider : public Ival_box { /* ... */ };

class Ival_dial : public Ival_box { /* ... */ };

class Flashing_ival_slider : public Ival_slider { /* ... */ };

class Popup_ival_slider : public Ival_slider { /* ... */ };
```

hoặc bằng đồ thị:



21.2.2 Interface Inheritance

[1]Hệ thống giao diện người dùng phải là một chi tiết triển khai bị ẩn với những người dùng không muốn biết về nó.

[2] Lớp Ival_box không được chứa dữ liệu.

[3] Không cần biên dịch lại mã sử dụng nhóm lớp Ival_box sau

thay đổi của hệ thống giao diện người dùng.

[4] Ival_boxes cho các hệ thống giao diện khác nhau sẽ có thể cùng tồn tại trong chương trình của chúng tôi.

Ở đây, tôi trình bày một bản đồ rõ ràng sang ngôn ngữ C ++.

Đầu tiên, tôi chỉ định lớp Ival_box làm giao diện thuần túy:

```
class Ival_box {
public:
        virtual int get_value() = 0;
        virtual void set_value(int i) = 0;
        virtual void reset_value(int i) = 0;
        virtual void prompt() = 0;
        virtual bool was_chang ed() const = 0;
        virtual ~Ival_box() { }
};
```

Điều này rõ ràng hơn nhiều so với khai báo ban đầu của Ival_box. Dữ liệu đã biến mất và các triển khai đơn giản của các hàm thành viên cũng vậy. Gone cũng là phương thức khởi tạo, vì không có dữ liệu nào để nó khởi tạo. Thay vào đó, tôi đã thêm một "hàm thuần ảo" để đảm bảo việc định nghĩa lại dữ liệu phù hợp sẽ được xác định trong các lớp dẫn xuất.

Định nghĩa của Ival_slider có thể giống như sau:

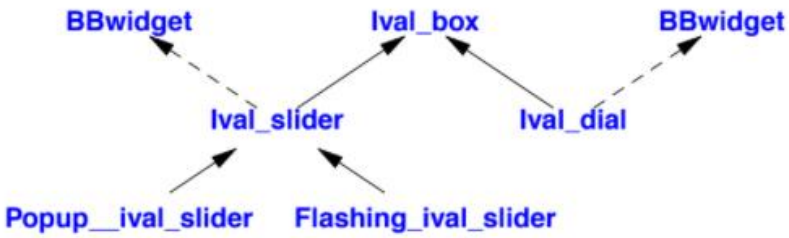
```
class Ival_slider : public Ival_box, protected BBwidget {
public:
        Ival_slider(int,int);
        ~Ival_slider() override;
        int get_value() override;
        void set_value(int i) override;
        // ...
protected:
        // ... các hàm ghi đè các hàm ảo BBwidget
        // ví dụ: BBwidget :: draw (), BBwidget :: mouse1hit () ...
private:
        // ... dữ liệu cần thiết cho slider ...
};
```

Hệ thống phân cấp Ival_box hiện có thể được định nghĩa như sau:

```
class Ival_box { /* ... */ };
```

```
class Ival_slider
        : public Ival_box, protected BBwidget { /* ... */ };
class Ival_dial
        : public Ival_box, protected BBwidget { /* ... */ };
class Flashing_ival_slider
        : public Ival_slider { /* ... */ };
class Popup_ival_slider
        : public Ival_slider { /* ... */};
```

bằng đồ thị:



Thiết kế này sạch hơn và dễ bảo trì hơn thiết kế truyền thống - và không kém phần hiệu quả. Tuy nhiên, nó vẫn không giải quyết được vấn đề kiểm soát phiên bản:

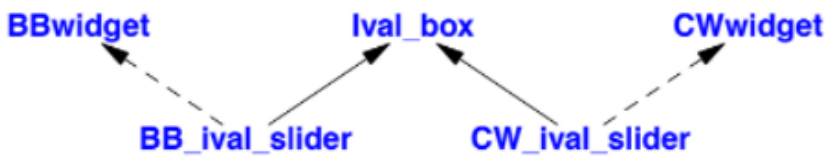
```
class Ival_box { /* ... */ }; // common
class Ival_slider
        : public Ival_box, protected BBwidget { /* ... */ }; // for BB
class Ival_slider
        : public Ival_box, protected CWwidget { /* ... */ }; // for CW
// …
```

Không có cách nào để Ival_slider cho BBwidgets cùng tồn tại với Ival_slider cho CWwidgets, ngay cả khi hai hệ thống giao diện người dùng có thể cùng tồn tại. Giải pháp rõ ràng là xác định một số lớp Ival_slider khác nhau với các tên riêng biệt:

```
class Ival_box { /* ... */ };
class BB_ival_slider
        : public Ival_box, protected BBwidget { /* ... */ };
class CW_ival_slider
        : public Ival_box, protected CWwidget { /* ... */ };
```

```
// …
```

bằng đồ thị:



21.2.3.1 Critique

Kết luận hợp lý của dòng suy nghĩ này là một hệ thống được đại diện cho người dùng như một hệ thống phân cấp của các lớp trừu tượng và được thực hiện bởi một hệ thống phân cấp cổ điển. Nói cách khác:

• Sử dụng các lớp trừu tượng để hỗ trợ kế thừa giao diện (§3.2.3, §20.1)

• Sử dụng các lớp cơ sở với việc triển khai các hàm ảo để hỗ trợ kế thừa thực thi (§3.2.3, §20.1).

21.2.4 Localizing Object Creation(Bản địa hóa Tạo đối tượng)

```
class Ival_maker {
        public:
        virtual Ival_dial* dial(int, int) =0; // make dial
        virtual Popup_ival_slider* popup_slider(int, int) =0; // make popup slider
        // ...
};
```

Bây giờ chúng tôi đại diện cho mỗi hệ thống giao diện người dùng bằng một lớp bắt nguồn từ Ival_maker:

```
class BB_maker : public Ival_maker { // make BB versions
public:
        Ival_dial* dial(int, int) override;
        Popup_ival_slider* popup_slider(int, int) override;
        // ...
};
class LS_maker : public Ival_maker { // make LS versions
public:
```

```
        Ival_dial* dial(int, int) override;
        Popup_ival_slider* popup_slider(int, int) override;
        // ...
};
```

Mỗi chức năng tạo ra một đối tượng của giao diện và kiểu triển khai mong muốn. Ví dụ:

```
Ival_dial* BB_maker::dial(int a, int b)
{
        return new BB_ival_dial(a,b);
}
        Ival_dial* LS_maker::dial(int a, int b)
{
        return new LS_ival_dial(a,b);
}
```

Với Ival_maker, người dùng hiện có thể tạo các đối tượng mà không cần biết chính xác hệ thống giao diện người dùng nào được sử dụng. Ví dụ:

```
void user(Ival_maker& im)
{
        unique_ptr<Ival_box> pb {im.dial(0,99)}; // create appropriate dial
// ...
}
BB_maker BB_impl; // for BB users
LS_maker LS_impl; // for LS users
void driver()
{
        user(BB_impl); // use BB
        user(LS_impl); // use LS
}
```

21.3 Multiple Inheritance

21.3.1 Multiple Interfaces(Nhiều giao diện)

Việc sử dụng nhiều lớp trừu tượng làm giao diện gần như phổ biến trong các thiết kế hướng đối tượng

(bằng bất kỳ ngôn ngữ nào có khái niệm về giao diện).

21.3.2 Multiple Implementation Classes

Bây giờ chúng ta có thể định nghĩa một lớp satellites truyền thông mô phỏng, lớp Comm_sat:

```
class Comm_sat : public Satellite, public Displayed {
public:
    // ...
};
```

bằng đồ thị:

Ngoài bất kỳ hoạt động nào được định nghĩa cụ thể cho một Comm_sat, sự kết hợp của các hoạt động

trên "Satellite " và " Displayed " có thể được áp dụng. Ví dụ:

```
void f(Comm_sat& s)
{
    s.draw(); // Displayed::draw()
    Pos p = s.center(); // Satellite::center()
    s.transmit(); // Comm_sat::transmit()
}
```

Tương tự, một Comm_sat có thể được chuyển cho một hàm Satellite và cho một hàm Displayed. Ví dụ:

```
void highlight(Displayed*);
Pos center_of_gravity(const Satellite*);
void g(Comm_sat* p)
```

```
{
    highlight(p); // chuyển một con trỏ đến phần Displayed của Comm_sat
    Pos x = center_of_gravity(p); // chuyển một con trỏ đến phần Satellite
    của Comm_sat
}
```

Việc thực hiện điều này rõ ràng liên quan đến một số kỹ thuật biên dịch (đơn giản) để đảm bảo rằng các chức năng Satellite nhìn thấy một phần khác của Comm_sat so với các chức năng Displayed. Các chức năng hàm ảo hoạt động như bình thường. Ví dụ:

```
class Satellite {
public:
    virtual Pos center() const = 0; // thuần ảo
    // ...
};
class Displayed {
public:
    virtual void draw() = 0;
    // ...
};
class Comm_sat : public Satellite, public Displayed {
public:
    Pos center() const override; // overr ide Satellite::center()
    void draw() override; // overr ide Displayed::draw()
    // ...
};
```

21.3.3 Ambiguity Resolution(Độ phân giải mơ hồ)

Hai lớp cơ sở có thể có các hàm thành viên trùng tên. Ví dụ:

```
class Satellite {
public:
    virtual Debug_info get_debug();
    // ...
```

```
};
class Displayed {
public:
        virtual Debug_info get_debug();
        // ...
};
```

Khi sử dụng Comm_sat, các chức năng này phải được phân loại. Điều này có thể được thực hiện đơn giản bằng cách đặt tên thành viên đủ điều kiện theo tên lớp của nó:

```
void f(Comm_sat& cs)
{
        Debug_info di = cs.g et_debug(); // error : ambiguous
        di = cs.Satellite::get_debug(); // OK
        di = cs.Displayed::g et_debug(); // OK
}
```

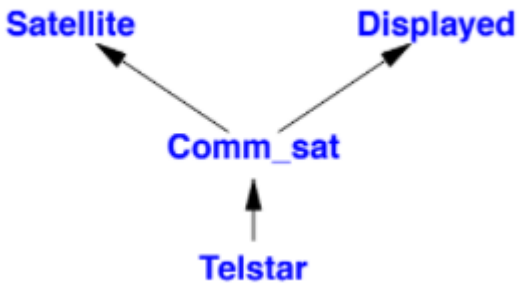
Tuy nhiên, lúc gọi hàm dễ bị chọn lộn vào hàm của lớp cơ sở, vì vậy tốt nhất là giải quyết các vấn đề như vậy bằng cách xác định một hàm mới trong lớp dẫn xuất:

```
class Comm_sat : public Satellite, public Displayed {
public:
        Debug_info get_debug() // overr ide Comm_sat::get_debug() and
        Displayed::get_debug()
        {
                Debug_info di1 = Satellite::get_debug();
                Debug_info di2 = Displayed::g et_debug();
                return merge_info(di1,di2);
        }
        // ...
};
```

Trong việc thực hiện chồng hàm, thường cần phải xác định rõ ràng tên để có được phiên bản phù hợp từ một lớp cơ sở. Một tên đủ điều kiện, chẳng hạn như Telstar :: draw, có thể đề cập đến một trận hòa được khai báo trong Telstar hoặc trong một trong các lớp cơ sở của nó. Ví dụ:

```
class Telstar : public Comm_sat {
public:
    void draw()
    {
        Comm_sat::draw(); // finds Displayed::draw
        // ... own stuff ...
    }
    // ...
};
```

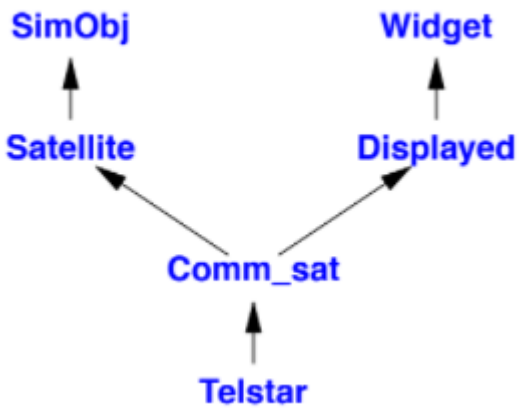
Đồ thị:



Sử dụng hàm ảo vào các lớp dẫn xuất:

```
class Satellite : public SimObj {
public:
    virtual Debug_info get_debug(); // gọi SimObj :: DBinf () và giải nén thông tin
    // ...
};
class Displayed : public Widget {
public:
    virtual Debug_info get_debug(); // đọc dữ liệu Widget và soạn Debug_info
    // ...
};
```

Đồ thị:




```
class Window {
public:
    void draw(); // display image     gọi hàm ảo draw trong class Window
    // ...
};
class Cowboy {
public:
    void draw(); // pull gun from holster     gọi hàm ảo draw trong class Cowboy
    // ...
};
class Cowboy_window : public Cowboy, public Window {       kế thừa nhiều lớp
    // ...
};
```

21.3.4 Repeated Use of a Base Class(Sử dụng lặp lại một lớp cơ sở)

```
struct Storable { // persistent storage
virtual string get_file() = 0;
virtual void read() = 0;
```

```
        virtual void write() = 0;
        virtual ~Storable() { }
        };
        class Transmitter : public Storable {
        public:
                void write() override;
                // ...
        };
        class Receiver : public Storable {
        public:
                void write() override;
                // ...
        };
        class Radio : public Transmitter, public Receiver {
        public:
                string get_file() override;
                void read() override;
                void write() override;
                // ...
        };

void Radio::write()
{
        Transmitter::write();
        Receiver::write();
        // ... write radio-specific infor mation ...
}
```

21.3.5 Virtual Base Classes(Lớp cơ sở ảo)

Điều gì sẽ xảy ra nếu Storable đã giữ dữ liệu và điều quan trọng là nó không được sao chép? Ví dụ: chúng ta có thể định nghĩa Storable để giữ tên của tệp được sử dụng để lưu trữ đối tượng:

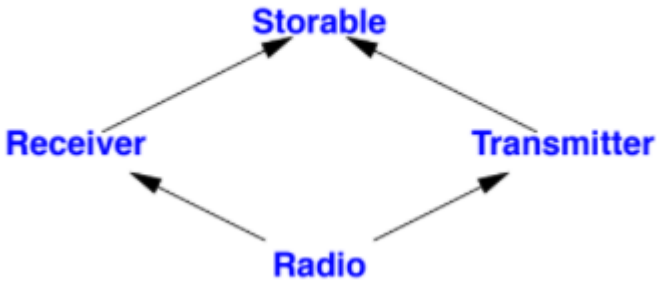
```
class Storable {
public:
    Storable(const string& s); // lưu trữ trong tệp có tên s
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
protected:
    string file_name;
    Storable(const Storable&) = delete;
    Storable& operator=(const Storable&) = delete;
};
```

Kế thừa ảo là một kỹ thuật C ++ đảm bảo rằng chỉ một bản sao của các biến thành viên của lớp cơ sở được kế thừa bởi các dẫn xuất cấp hai

```
class Transmitter : public virtual Storable {
public:
    void write() override;
    // ...
};
class Receiver : public virtual Storable {
public:
    void write() override;
    // ...
};
class Radio : public Transmitter, public Receiver {
public:
    void write() override;
    // ...
```

};

Bằng đồ thị:



21.3.5.1 Constructing Virtual Bases(Xây dựng cơ sở ảo)

```
struct V {
    V(int i);
    // ...
};
struct A {
    A(); //hàm khởi tạo mặc định A
    // ...
};
struct B : virtual V, vir tual A {
    B() :V{1} { /* ... */ }; //hàm khởi tạo mặc định B; phải khởi tạo cơ sở V
    // ...
};
class C : virtual V {
public:
    C(int i) : V{i} { /* ... */ }; phải khởi tạo cơ sở V
    // ...
};
class D : virtual public B, virtual public C {
    // ngầm định lấy cơ sở ảo V từ B và C
```

```
        // ngầm nhận cơ sở ảo A từ B
public:
        D() { /* ... */ } // error: không có hàm tạo mặc định cho C hoặc V
        D(int i) :C{i} { /* ... */ }; // error: không có hàm tạo mặc định cho V
        D(int i, int j) :V{i}, C{j} { /* ... */ } // OK
        // ...
};
```

21.3.5.2 Calling a Virtual Class Member Once Only(Gọi một thành viên lớp ảo Chỉ một lần)

```
class Window {
public:
        // basic stuff
        virtual void draw();
};
```

Ngoài ra, có nhiều cách khác nhau:

```
class Window_with_border : public virtual Window {
        // border stuff
protected:
        void own_draw(); // display the border
public:
        void draw() override;
        };
class Window_with_menu : public virtual Window {
        // menu stuff
protected:
        void own_draw(); // display the menu
public:
```

```
        void draw() override;
    };
```

Các hàm own_draw () không cần phải ảo bởi vì chúng được gọi từ bên trong một hàm draw () ảo " biết " loại đối tượng mà nó được gọi.

Từ đó, chúng ta có thể tạo một lớp Đồng hồ hợp lý:

```
    class Clock : public Window_with_border, public Window_with_menu {
        // clock stuff
    protected:
        void own_draw(); // display the clock face and hands
    public:
        void draw() override;
    };
```

Bằng đồ thị: