Item 35: Consider alternatives to virtual functions

فرض کنید که بر روی یک بازی ویدیویی کار می‌کنید، و قصد دارید که یک ساختار برای کاراکترهای بازی ایجاد کنید. در این بازی این که کاراکترها صدمه ببینند یک چیز طبیعی است، به عبارت دیگر شاهد کاهش سلامتی بازیکنان باشیم. بنابراین قصد دارید که یک تابع عضو به نام healthValue پیشنهاد بدهید، که یک عدد صحیح برمیگرداند که این عدد نشان‌دهنده‌ی میزان سلامتی هر کاراکتر است. از آنجایی که میزان سلامتی هر کاراکتر، ممکن است به شیوه‌های متفاوتی محاسبه شود، در این صورت تابع healthValue را به صورت virtual تعریف کرده‌اید:

```
class GameCharachter{
public:
   virtual int healthValue() const;   //return charachter's health rating,
                                      //derived classes may redefine this
};
```

این حقیت که healthValue به صورت pure virtual اعلان نشده است، پیشنهاد می‌دهد که یک الگوریتم پیش‌فرض برای محاسبه‌ی سلامتی وجود دارد (آیتم ۳۴ را ببینید).

این واضح‌ترین حالت ممکن برای طراحی است. از آنجایی که این طراحی خیلی واضح است، ممکن است که دید کافی از سایر روش‌های جایگزین را از شما بگیرد. در آیتم قصد داریم که به شما کمک کنیم تا از روش‌هایی به غیر از طراحی شیءگرا این مساله را حل کنید.

The Template Method Pattern via the Non-Virtual Interface Idiom

ما با کلاس فکری‌ای شروع می‌کنیم که پیشنهاد می‌کند، توابع virtual همواره باید به صورت private تعریف شوند. طرفداران این کلاس فکری پیشنهاد می‌دهند که یک طراحی خوب به این صورت است که تابع healthValue را به صورت عضو public نگه داریم ولی آن را به صورت non-virtual تعریف کنیم و یک فراخوانی private به تابع virtual که کار اصلی را انجام میدهد داشته باشیم، یعنی تابع doHealthValue :

```
class GameCharachter{
public:
   int healthValue() const
   {
      ....                              //do before stuff
      int retVal=doHealthValue();
      ....                              //do after stuff
      return retVal;
   }
```

```
private:
   virtual int doHealthValue() const
   {
      ....
   }
};
```

در این کد(و در کل این آیتم)، من بدنه‌ی توابع عضو را در تعریف کلاس می‌نویسم. همچنانکه آیتم ۳۰ نشان خواهد داد، به شکلی غیر صریح این باعث اعلان iniline آن‌ها می‌شود. من تنها به خاطر این که فهم قضیه آسان‌تر شود این کار را انجام خواهم داد و قصد دیگری ندارم.

این طراحی ساده(یعنی فراخوانی توابع private virtual که به صورت غیر مستقیم از طریق توابع عضو public non-virtual صورت می‌پذیرد) به عنوان روش non-virtual interface یا NVI شناخته می‌شود. این طراحی یک حالت کلی‌تر از طراحی‌ای است که Template Method نامیده می‌شود(این طراحی ربطی به templates های C++ ندارد). من توابع non-virtual (یعنی healthValue) را توابع virtual پوششی می‌نامم.

یک مزیت از طراحی NVI به خاطر دو کامنت do before stuff و do after stuff است که در کد مشخص شده است. این کامنت‌ها مشخص‌کننده‌ی کدی هستند که حتما قبل و بعد از تابع virtual فراخوانی می‌شوند. این بدین معنی است که این پوشش اجازه میدهد که چیزهای لازم قبل از تابع آماده و تنظیم شوند، و بعد از این که این تابع فراخوانی شد، این چیزها پاک شوند. به طور مثال، در before stuff می‌توانیم یک mutex را lock کنیم، و یک چیزی را در log ذخیره کنیم. و در after stuff می‌توانیم mutex مربوطه را unlock کنیم. در حقیقت وقتی اجازه میدهیم که کاربر توابع virtual را مستقیما فراخوانی کند، راه بهتری برای انجام این کار وجود ندارد.

ممکن است این فکر به ذهن شما خطور کند که روش NVI به گونه‌ای است که کلاس‌های مشتق شده باید توابع private virtual را دوباره تعریف کنند(دوباره نویسی توابعی که نمی‌توانند فراخوانی کنند!!.) در واقع این هیچگونه تناقض طراحی ندارد. دوباره‌نویسی توابع virtual به طور خاص مشخص می‌کند که این توابع باید به چه صورت کار کند. فراخوانی virtual function مشخص می‌کند که چه موقع آن انجام می‌شود. این مسایل مستقل بوده و به هم ارتباطی ندارد. روش NVI اجازه میدهد که کلاس‌های مشتق شده توابع virtual را دوباره‌نویسی کنند، بنابراین به آن‌ها اجازه داده می‌شود که کنترل کاملی بر روی توابعی که پیاده می‌شود، داشته باشند، ولی کلاس base این حق را برای خودش حفظ می‌کند که چه موقع تابع فراخوانی شود. این ممکن است در نگاه اول عجیب به نظر برسد، ولی این قانون C++ که کلاس‌های مشتق شده می‌توانند توابع private virtual را دوباره‌نویسی کنند کاملا منطقی است.

تحت روش NVI ، حتما این‌گونه نیست که توابع virtual باید به صورت private باشد. در برخی ساختار کلاس‌ها، پیاده‌سازی کلاس‌های مشتق شده از توابع virtual به این صورت است که این انتظار وجود دارد که بتوانند همتای خود را در کلاس base صدا بزنند، و برای این که چنین فراخوانی درست باشد، virtualها باید به صورت protected باشند نه private . در برخی موارد توابع virtual حتما باید public باشند(به طور مثال، مخرب‌ها در کلاس base چندریختی – آیتم ۷ را ببینید)، ولی در این مورد روش NVI دیگر قابل اعمال نیست.

استراتژی بر اساس اشاره‌گر به توابع

روش NVI یک جایگزین جالب برای توابع public virtual هستند، ولی از نقطه نظر طراحی، این بیشتر شبیه به پوشاندن یک پنجره است. و در حقیقت، همچنان ما داریم از توابع virtual برای محاسبه‌ی سلامت کاراکتر استفاده می‌کنیم. یک طراحی جالب‌تر ایت است که بگیم محاسبه‌ی سلامتی هر کاراکتر مستقل از نوع کاراکتر بوده (چنین محاسباتی مستلزم این است که این محاسبات جزیی از کاراکتر باشد). به طور مثال، می‌توانیم کاری کنیم که سازنده‌ی هر کاراکتر به صورت یک اشاره‌گر به تابع محاسبه‌ی سلامتی پاس داده شوند، و از آن تابع بخواهیم که محاسبات واقعی را انجام دهد.

```
class GameCharacter;    //forward declaration
//function for the default health calculation algorithm
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter{
public:
   typedef int (*HealthCalcFunc)(const GameCharacter&);
   explicit GameCharacter(HealthCalcFunc hcf=defaultHealthCalc):healthFunc(hcf)
   {   }

   int healthValue() const
   {
      return healthFunc(*this);
   }
private:
   HealthCalcFunc healthFunc;
};
```

این رویکرد یک کاربرد ساده از یک استراتژی دیگر است. نسبت به روش‌هایی که مبتنی بر توابع مجازی هستند، این روش انعطاف‌پذیری جالبی را به دست می‌دهد:

- نمونه‌های متفاوت از کارکترهای یکسان می‌توانند توابع متفاوتی برای محاسبه‌ی سلامتی داشته باشند. به طور مثال:

```
class EvilBadGuy:public GameCharacter{
public:
   explicit EvilBadGuy(HealthCalcFunc hcf=defaultHealthCalc):GameCharacter(hcf){}

};

int loseHealthQuickly(const GameCharacter&);
int loseHealthSlowly(const GameCharacter&);

EvilBadGuy ebg1(loseHealthQuickly);
EvilBadGuy ebg2(loseHealthSlowly);
```

- محاسبه‌ی سلامتی برای یک کاراکتر خاص ممکن است که در لحظه‌ی اجرا تغییر کند. به طور مثال، GameCharachter ممکن است که یک تابع عضو به نام setHealthCalculator را پیشنهاد بدهد، که اجازه‌ی تغییر تابع تعیین سلامتی را بدهد.

از سوی دیگر، از آنجایی که تابع محاسبه‌ی سلامتی، یک تابع عضو از کلاس GameCharachter نیست، بدین معنی است که تابع، دسترسی خاصی به اعضای داخلی شیءای که محاسبات آن را انجام می‌دهد ندارد. به طور مثال، defaultHealthCalc دسترسی‌ای به قسمت‌های non-public کلاس EvilBadGuy ندارد. اگر سلامتی یک کاراکتر را بتوان به طور کامل بر اساس اطلاعات موجود در رابط public کاراکتر انجام داد، هیچ مشکلی وجود نخواهد داشت. در حقیقت، این امکان وجود دارد که شما با جایگزینی تابع داخل کلاس(به طور مثال، از طریق تابع عضو) با تابع بیرون از کلاس (به طور مثال، یک تابع غیر عضو با غیر دوست و یا از طریق یک تابع غیر دوست عضو از یک کلاس دیگر) به مشکل بخورید. این مشکل، برای بقیه‌ی آیتم نیز برجا خواهد بود، چرا که همه‌ی طراحی‌های دیگری که بررسی خواهیم کرد، تابع را بیرون از ساختار GameCharachter خواهیم آورد.

به عنوان یک قاعده‌ی کلی، تنها راه برای برطرف کردن نیازی که توابع غیر عضو به قسمت‌های non-public دارند این است که کپسوله‌سازی کلاس رو ضعیف کنیم. به طور مثال، کلاس ممکن است که تابع غیر عضو را به عنوان دوست اعلان کند، یا دسترسی عمومی به برخی از پیاده‌سازی‌ها بدهد. در هر صورت برای داشتن مزیت‌هایی که استفاده از اشاره‌گر به تابع دارد(یعنی، توانایی داشتن تابع مربوط محاسبه‌ی سلامتی به ازای هر شیء در هنگام اجرا)، ممکن است که میزان کپسوله‌سازی کلاس را کاهش دهیم، چیزی که ممکن است شما را از بکاربردن این طراحی منصرف کند.