Item 38: Model “has-a” or “is-implemented-in-terms-of” through composition

ترکیب یا composition به رابطه‌ی بین چند تایپ گفته می‌شود، که وقتی رخ میدهد که اشیاء از یک نوع، اشیاء از نوع دیگری را شامل باشند. به طور مثال:

```
#include <iostream>
class Address{};
class PhoneNumber{};
class Person{
public:

private:
    std::string name;                    //composed object
    Address address;                     //ditto
    PhoneNumber voiceNumber;             //ditto
    PhoneNumber faxNumber;               //ditto
};
```

در این مثال، اشیاء Person متشکل شده از اشیاء Adress ، string و PhoneNumber هستند. در میان برنامه‌نویسان، لغت ترکیب یا composition دارای مترادف‌های بسیاری است. ترکیب همچنین تحت عنوان aggregation ، containment ، layering و embedding نیز شناخته می‌شود.

در آیتم ۳۲ توضیح داده شد که معنای وراثت عمومی is-a می‌باشد. ترکیب نیز دارای یک معنای مخصوص به خود است. در واقع، ترکیب دارای دو معنا است. ترکیب یا به معنای "دارای" و یا "پیاده‌سازی شده در " می‌باشد. و دلیل آن این است که شما در دو ناحیه‌ی متفاوت برنامه‌نویسی کار می‌کنید. برخی از اشیاء در برنامه‌های شما مطابق اشیاء در مدلی واقعی بوده، مانند مردم، وسایل نقلیه، فریم‌های یک ویدیو و غیره. چنین اشیایی جزوی از **ناحیه‌ی برنامه** می‌باشد. اشیاء دیگر به طور کامل مصنوعی و ساختگی می‌باشد، مثل بافرها، mutex ها ، درخت‌های جستجو و مواردی از این قبیل. چنین اشیایی در واقع جزوی از **ناحیه‌ی پیاده‌سازی برنامه** می‌باشد. وقتی که در ناحیه‌ی برنامه، بین اشیاء **ترکیب** اتفاق می‌افتد، این ترکیب بیانگر ارتباط has-a خواهد بود. و وقتی که در ناحیه‌ی پیاده‌سازی اتفاق بیفتد، بیانگر رابطه‌ی پیاده‌سازی شده در میباشد.

کلاس Person در مثال قبلی بیانگر رابطه has-a می‌باشد. شیء Person دارای یک نام، دارای یک آدرس، شماره تماس صوتی و فکس می‌باشد. ما هرگز نمیگیم که یک شخص همون اسمشه(رابطه‌ی is-a) و یا شخص همون شماره تماس. بلکه میگیم که شخص دارای یک اسم و یک آدرس

می‌باشد(رابطه‌ی has-a). بیشتر مردم با این دو مفهوم مجزا مشکلی ندارند، بنابراین گیج شدن سر این قاعده‌ی is-a و has-a به ندرت اتفاق می‌افتد.

چیزی که برای بیشتر افراد مشکل ایجاد می‌کند، تفاوتی است که بین is-a و is-implemented-in-terms-of است. به طور مثال، فرض کنید که شما به یک template برای نشان دادن مجموعه‌ای از اشیاء کوچک نیاز دارید، یعنی یک مجموعه بدون دوباره‌نویسی کد. از اونجایی که reuse خیلی چیز خوبی است، به طور غریزی ما از set template کتابخانه‌ی استاندارد استفاده می‌کنیم. چرا یک template جدید بنویسیم، در حالی که قبلا یکی نوشته شده؟

متاسفانه، پیاده‌سازی set معمولا منجر به سربار سه اشاره‌گر به ازای هر المان می‌شود. چرا که set تحت درخت‌های جستجو متعادل پیاده‌سازی شده، چیزی که به اونا اجازه بده که جستجوی لگاریتمی، الحاق یا insertions و پاک کردن رو بده. وقتی که سرعت از فضا مهم‌تر باشد. تابع set کتابخانه‌ی استاندارد گزینه‌ی مناسبی برای شما نخواهد بود. بنابراین به نظر میرسد که ما نیاز داریم که template خودمان را بنویسیم.

با این وجود، هنوز reuse بودن خیلی به درد بخور است. اگر متخصص ساختمان داده باشید، می‌دانید که گزینه‌های زیادی برای پیاده‌سازی مجموعه‌ها داریم، یکی این است که از linked list ها استفاده کنیم. همچنین می‌دانیم که کتابخانه‌ی استاندارد C++ یک template برای list دارد، بنابراین ما تصمیم میگیریم که از آن استفاده کنیم.

به طور خاص، شما تصمیم میگیرید که template set در حال تولد شما از list ارث‌بری کند. بنابراین

set <T>

از

list<T>

ارث‌بری می‌کند. در نهایت، در پیاده‌سازی شما یک شیء set یک شیء list خواهد بود. بنابراین تصمیم میگیرید که set template شما به صورت زیر باشد:

```
template <typename T>
class Set:public std::list<T>    //the wrong way to use list for Set
{

};
```

در اینجا همه چیز ممکن است درست به نظر برسد، ولی در حقیقت یک چیزی کاملا اشتباه است. همچنانکه در آیتم ۳۲ توضیح داده شد، اگر D یک B باشد، هر چیزی که در مورد B درست باشد، برای

D نیز درست است. در صورتی که، یک list ممکن است که دارای duplicate باشد، بنابراین اگر مقدار ۳۰۵۱ دوبار وارد list شود، این لیست دارای دو کپی از ۳۰۵۱ خواهد بود. در مقابل، یک Set نمی‌تواند duplicate داشته باشد، بنابراین اگر مقدار ۳۰۵۱ دوبار وارد ۳۰۵۱ شود، Set تنها یک کپی از آن تهیه می‌کند. بنابراین این واقعیت که set یک list است درست نیست، چرا که برخی چیزهایی که برای list درست است برای Set درست نیست.

به خاطر این که ارتباط بین این دو کلاس از قاعده‌ی is-a پیروی نمی‌کند، ارث‌بری عمومی یک مدل اشتباه می‌باشد. راه درست برای این کار این است که شیء Set را بتوان با استفاده از list پیاده‌سازی کرد:

```
class Set
{
public:
    bool member(const T& item) const;

    void insert(const T& item);
    void remove(const T& item);

    std::size_t size() const;
private:
    std::list<T> rep;     //representation for Set data
};
```

توابع عضو Set را می‌توان بیشتر با استفاده از list و برخی قسمت‌های دیگر کتابخانه استاندارد پیاده‌سازی کرد، بنابراین پیاده‌سازی پیچیده نخواهد بود، البته این مستلزم این است که با مبنای برنامه‌نویسی STL آشنایی داشته باشید:

```
template<class T>
bool Set<T>::member(const T &item) const
{
    return std::find(rep.begin(),rep.end(),item) != rep.end();
}
template<class T>
void Set<T>::insert(const T &item)
{
    if(!member(item)) rep.push_back(item);
}
```

```
template<class T>
void Set<T>::remove(const T &item)
{
    typename std::list<T>::iterator it=          //see Item 42 for Info on "typename" in here
            std::find(rep.begin(),rep.end(),item);
    if(it!=rep.end()) rep.erase(it);
}
template<class T>
size_t Set<T>::size() const
{
    return rep.size();
}
```

این توابع به اندازه‌ی کافی ساده هستند که آن‌ها را یک کاندید مناسب برای inline کردن می‌کنند، البته می‌دانم که قبل از این تصمیم شما یک مرور روی مباحث آیتم ۳۰ خواهید انداخت.

یک نفر ممکن است بحث کند که رابط Set می‌تواند خیلی بیشتر مطابق با آیتم ۱۸ باشد که در آنجا بحث روی این بود که رابط را به گونه‌ای بنویسیم که برای استفاده درست آسان و برای استفاده اشتباه دشوار باشد. ولی پیروی کردن از توافقاتی که در STL وجود دارد منجر به افزودن خیلی چیزهای دیگر به Set می‌شود که ممکن است رابطه‌ی بین آن و list را از مسدود کند. از آنجایی که این رابطه، هدف این آیتم بود، ما به compatibility با STL نپرداختیم.

Item 39: Use private inheritance judiciously

آیتم ۳۲ نشان داد که C++ وراثت عمومی را یک رابطه‌ی is-a می‌داند. C++ این را با استفاده از کامپایلر نشان میدهد، وقتی در یک ساختار که کلاس Student به صورت عمومی از کلاس Person ارث‌بری کرده، کامپایلر وقتی نیاز باشد به طور غیر صریح می‌تواند اشیاء Student را به اشیاء Person تبدیل کند و به فراخوان توابع بدهد. قسمتی از این مثال را به جای استفاده از وراثت عمومی، با استفاده از وراثت خصوصی با همدیگر می‌بینیم.

```
class Person{};
class Student:private Person{};          //inheritance is now private
void eat(const Person& p);               //anyone can eat
void study(const Student& s);            //only student study
Person p;                                //p is a Person
Student s;                               //s is a Student
```

```
eat(p);   //fine,p is a Person
eat(s);   //error! a Student isn't a Person
```

واضح است که وراثت خصوصی به معنای is-a نیست. این به چه معناست؟

خب بیایید رفتار وراثت خصوصی را بیشتر بررسی کنیم. خب، اولین قاعده‌ای که در مورد وراثت خصوصی دیدیم این بود که بر خلاف وراثت عمومی، در صورتی که وراثت به صورت خصوصی باشد کامپایلرهای عموما کلاس مشتق شده (مثل کلاس Student) را به کلاس پایه (مثل Person) تبدیل نمی‌کنند. به همین خاطر فراخوانی eat برای شیء s کار نمی‌کند. قاعده‌ی دوم این است که اعضایی که از کلاس پایه خصوصی به ارث برده می‌شوند برای کلاس مشتق شده به صورت private می‌شوند، حتی اگر آن‌ها در کلاس پایه به صورت protected و یا public باشند.

این رفتارها برای ما یک معنا خواهند داشت. ارث‌بری عمومی به معنای is-implemented-in-terms-of خواهد بود. اگر شما یک کلاس به نام D بسازید که به صورت خصوصی از کلاس B ارث‌بری کند، شما این کار را انجام میدهید چون می‌خواهید از برخی ویژگی‌های موجود در B که به آن‌ها علاقه‌مند هستید استفاده کنید. بنابراین، وراثت خصوصی یک تکنیک پیاده‌سازی بوده(به همین خاطر است که هر چیزی که از کلاس پایه به صورت خصوصی به ارث می‌برید در کلاس شما private می‌شوند. با استفاده از چیزی که در آیتم ۳۴ بررسی کردیم، ارث‌بری خصوصی به معنای پیاده‌سازی‌ای است که باید به ارث برده شود. رابط‌ها باید نادیده گرفته شوند. اگر D به صورت خصوصی از B به ارث برده شود، این بدین معنی است که اشیاء D از منظر B پیاده‌سازی شده‌اند و هیچ معنای دیگری ندارد. وراثت خصوصی در پروسه‌ی طراحی نرم‌افزار هیچ معنای خاصی ندارد، تنها در هنگام پیاده‌سازی نرم‌افزار معنا پیدا می‌کند.

این حقیقت که وراثت عمومی به معنای is-implemented-in-terms-of است، ممکن است ذهن ما را آشفته کند، چرا که در آیتم ۳۸ دیدیم که ترکیب یا composition به همین معنا است. چطور قرار است که بین این‌ها انتخابی انجام دهیم؟ پاسخ ساده است: هر کجا که توانستید از ترکیب استفاده کنید، و تنها وقتی که واقعا راهکار دیگه‌ای نداشتید از ارث‌بری خصوصی استفاده کنید. چرا این طور است؟ وقتی که اعضای protected و یا توابع virtual وارد صحنه می‌شوند.

فرض کنید که بر روی یک برنامه که درگیر Widget ها هست کار می‌کنیم، و تصمیم میگیریم که نیاز داریم بدانیم چطور widget ها مورد استفاده قرار میگیرند. به طور مثال، نه تنها نیاز داریم بدانیم که توابع عضو Widget چه میزان مورد استفاده قرار میگیرند، بلکه می‌خواهیم بدانیم که نسبت این فراخوانی‌ها در طول زمان چطور تغییر می‌کند. برنامه‌ها با فازهای مجزای اجرایی می‌توانند رفتار متفاوتی در طی فازهای متفاوت داشته باشند. به طور مثال، توابعی که در طی فاز parsing کامپایلر مورد استفاده قرار میگیرد کاملا متفاوت از توابعی است که در طی بهینه‌سازی و یا تولید کد مورد استفاده قرار میگیرد.

در این صورت تصمیم میگیریم که کلاس Widget رو به نحوی تغییر بدهیم که تعداد فراخوانی‌های توابع عضو را بشماریم. در لحظه‌ی runtime ، ما این اطلاعات را متناوبا مورد آزمایش قرار میدهیم. برای این که این به طور مناسبی کار کند، ما نیاز داریم که یک تایمر تنظیم کنیم که بدانیم چه موقع باید اطلاعات مورد نیاز را جمع‌آوری کنیم.

از آنجایی که ترجیح ما این است که از کدهای موجود دوباره استفاده کنیم، داخل ابزارهایی که داریم یک جستجو انجام داده و کلاس زیر را پیدا می‌کنیم:

```
class Timer{
public:
    explicit Timer(int tickFrequncy);
    virtual void onTick() const;   //automatically called for each tick
};
```

این دقیقا چیزی است که ما به دنبال آن بودیم. یک Timer که بتوان با یک فرکانس خاص آن را تنظیم کرد، و در هر بار که به نقطه‌ی tick خود می‌رسد، یک تابع virtual را فراخوانی بکند. ما می‌توانیم این تابع virtual را خودمان دوباره‌نویسی کنیم تا شرایط Widget را در آن لحظه مورد بررسی قرار دهیم.

برای این که Widget بتواند تابع virtual را دوباره‌نویسی کند، Widget بایستی از Timer ارث‌بری کند. ولی استفاده از وراثت عمومی در این مورد مناسب نیست. چرا که Widget یک Timer نیست. کاربران Widget نباید این امکان را داشته باشند که onTick را بر روی Widget صدا بزنند، چرا که این تابع جزویی از مفهوم Widget نیست. این که اجازه بدهیم که چنین تابعی از روی Widget صدا زده شود، به این معنی است که به کاربر اجازه داده‌ایم که به راحتی از رابط ما اشتباه استفاده کند، که این کاملا مخالف توصیه‌ای است که در آیتم ۱۸ آورده‌ایم که رابط‌ها باید به گونه‌ای طراحی شوند که برای استفاده درست آسان و برای استفاده اشتباه سخت‌تر باشند. بنابراین استفاده از وراثت عمومی در اینجا یک گزینه‌ی صحیح نیست.

بنابراین ما از وراثت خصوصی استفاده می‌کنیم:

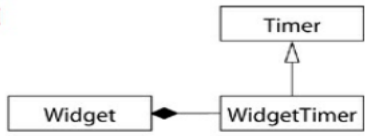
```
class Timer{
public:
    explicit Timer(int tickFrequncy);
    virtual void onTick() const;   //automatically called for each tick
};
class Widget: private Timer{
private:
    virtual void onTick() const;   //look at Widget usage data,etc.
};
```

با توجه به تقوایی که ارث‌بری خصوصی دارد، تابع onTick کلاس Timer برای Widget به صورت خصوصی می‌شود، و می‌توانیم با تعریف دوباره آن، نگهش داریم. مثل قبل، قرار دادن onTick در رابط عمومی ممکن است باعث شود کاربر به طور اشتباهی فکر کند که می‌تواند آن را فراخوانی کند، و این مخالف آیتم ۱۸ می‌باشد.

این یک طراحی خوب است، ولی ارزش این را دارد که اشاره کنیم ارث‌بری خصوصی لازم نیست. و اگر می‌خواستیم می‌توانستیم از composition به جای آن استفاده کنیم. ما می‌توانستیم یک کلاس خصوصی در درون Widget ایجاد کنیم که به صورت عمومی از Timer ارث‌بری کرده، و در آنجا onTick را دوباره تعریف می‌کردیم، و یک شیء از آن نوع را داخل Widget قرار میدادیم:

```
class Widget{
private:
   class WidgetTimer:public Timer{
     public:
       virtual void onTick() const;
   };

   WidgetTimer timer;
};
```



این طراحی خیلی پیچیده‌تر از این است که تنها از ارث‌بری خصوصی استفاده کنیم، چرا که هم درگیر ارث‌بری عمومی است و هم composition دارد، و همچنین یک کلاس جدید به نام WidgetTimer باید تعریف شود. اگر صادقانه بخواهم بگویم، تنها به این خاطر که یادآوری کنم، روش‌های زیادی برای طراحی وجود دارد و ارزش آن را دارد که روش‌های مختلف را خودتان تست کنید و یاد بگیرید(آیتم ۳۵ را هم ببینید). من می‌توانم دو دلیلی که چرا ممکن است شما ارث‌بری عمومی با composition را به ارث‌بری خصوصی ترجیح بدهید را حدس بزنم.

اول این که، می‌خواهید Widget را به گونه‌ای طراحی کنید که به کلاس‌های مشتق شده اجازه بدهید که onTick را داشته باشند، ولی نتوانند آن را دوباره‌نویسی کنند. اگر Widget از Timer ارث‌بری کرده باشد، این مورد امکان پذیر نیست، حتی اگر ارث‌بری به صورت خصوصی باشد(به خاطر بیاورید که در آیتم ۳۵ دیدیم که کلاس‌های مشتق شده می‌توانند توابع virtual را دوباره تعریف کنند، حتی اگر اجازه

نداشته باشند آن‌ها را صدا بزنند). اما اگر WidgetTimer در داخل Widget به صورت خصوصی باشد و از Timer ارث‌بری کرده باشد، کلاس‌های مشتق شده Widget هیچ دسترسی‌ای به WidgetTimer ندارند، بنابراین نمی‌توانند از آن ارث بری کنند و یا توابع virtual را دوباره تعریف کنند. اگر شما در Java و یا C# تجربه داشته باشید و دلتان برای این تنگ شده بود که کلاس‌های مشتق شده نتوانند توابع virtual را دوباره تعریف کنند(یعنی متد final در Java و Sealed در C#)، حال یک ایده برای این رفتار در C++ دارید.

دوم این که، می‌خواهید که وابستگی‌های Widget را تا حد امکان پایین نگه دارید. اگر Widget از Timer ارث‌بری بکند، تعریف Timer باید در هنگام کامپایل Widget موجود باشد، بنابراین در فایل Widget باید Timer.h را include بکنیم. به عبارت دیگر، اگر WidgetTimer را بیرون از Widget ببریم و Widget تنها شامل یک اشاره‌گر به WidgetTimer باشد، Widget می‌تواند با یک اعلان ساده به کلاس WidgetTimer کار کند و نیازی به #include کردن چیزی از Timer نیست. برای سیستم‌های بزرگ چنین مجزاسازی‌هایی خیلی مهم است(برای مشاهده‌ی جزییات کاهش وابستگی‌های کامپایل آیتم ۳۱ را ببینید.)

من نشان دادم که ارث‌بری خصوصی وقتی مفید است که بخواهیم کلاس‌های مشتق شده به قسمت‌های protected از یک کلاس پایه دسترسی داشته باشند، یا اینکه بخواهیم یک یا چند تابع virtual را دوباره تعریف کنیم، ولی به صورت مفهومی رابطه‌ی بین دو کلاس is-implemented-in-terms-of باشد نه is-a . اگر چه، من گفتم که یک مورد مرزی وجود دارد که ممکن است باعث شود که ارث‌بری خصوصی را به ترکیب دهید.

این مورد وقتی است که با یک کلاس سر و کار داریم که هیچ دیتایی درون آن وجود ندارد. چنین کلاس‌های هیچ داده‌ی عضو non-static ندارند، هیچ تابع مجازی ندارند(چرا که وجود چنین توابعی یک vptr به هر شیء اضافه می‌کند-آیتم ۷ را ببینید) و هیچ کلاس پایه‌ی virtual ای ندارند(چرا که چنین توابعی سایز سربار را افزایش میدهند-آیتم ۴۰ را ببینید). به صورت مفهومی، اشیاء از چنین کلاسی نباید هیچ حافظه‌ای بگیرند، چرا که هیچ داده‌ای به ازای شیء وجود ندارد که بخواهد ذخیره شود. اگر چه، در C++ دلایل فنی وجود دارد که کلاس‌های **مستقل** نباید سایزشون صفر باشه، در این صورت اگر شما یک همچین چیزی بنویسید:

```
class Empty{};  //has no data, so objects should,
                //use no memory
class HoldsAnInt{   //should need only space for an int
private:
    int x;
    Empty r;   //should require no memory
```

```
};
```

درخواهید یافت که sizeof(HoldsAnInt)>sizeof(int) می‌باشد. یک داده‌ی عضو Empry نیازمند حافظه است. در بیشتر کامپایلرها sizeof(Empty) برابر با ۱ خواهد بود، چرا که C++ مخالف صفر بودن سایز اشیاء **مستقل** بوده و معمولا در پشت پرده یک char به اشیاء empty اضافه می‌کند. اگر چه، ضرورت‌های مرتب‌سازی(آیتم ۵۰) ممکن است که باعث شود یک حاشیه به کلاس‌هایی مثل HoldAnInt اضافه شود، بنابراین HoldAnInt ممکن است که بیشتر از char شوند، و سایز آن‌ها آنقدری بزرگ شود که در یک Int جا بگیرند.(در همه‌ی کامپایلرهایی که من تست کردم، این چیزی بود که اتفاق افتاد).

احتمالا متوجه شده‌اید که من از واژه **مستقل** برای اشیایی که نباید اندازه‌ی صفر داشته باشند استفاده می‌کنم. این محدودیت برای قسمت‌های کلاس base ازکلاس‌های مشتق شده اعمال نمی‌شود، چرا که آن‌ها مستقل نیستند. اگر شما به جای این که یک شیء از نوع Empty داشته باشید، از Empty ارث‌بری کنید:

```
class HoldsAnInt:private Empty{
private:
    int x;
};
```

احتمالا درخواهید یافت که sizeof(HoldsAnInt)=sizeof(int) می‌باشد. این به base empty optimization یا EBO معروف است، و تمام کامپایلرها آن را پیاده‌سازی کرده‌اند. اگر شما یک توسعه‌دهنده‌ی کتابخانه باشید که کاربران شما حساسیت زیادی در مورد فضا داشته باشند، EBO ارزش دانستن را دارد. همچنین لازم است بدانیم که EBO تنها در حالت single inheritance عمل می‌کند. قوانین C++ می‌گوید که EBO تنها بر روی کلاس‌های مشتق شده‌ای که یک کلاس base دارند قابل اعمال است.

در عمل، کلاس‌های empty واقعا تهی نیستند. حتی اگر هیچ داده‌ی non-static نداشته باشند، اغلب شامل typedef ها، enum ها، اعضای داده‌ای static و یا توابع non-virtual هستند. STL دارای کلاس‌های خالی بسیاری است که توابع عضو مفیدی دارند(معمولا typedefs)، که شامل کلاس‌های پایه unary_function و binary_function می‌باشند. به خاطر پیاده‌سازی گسترده EBO ، چنین ارث‌بری‌هایی به ندرت موجب افزایش سایز کلاس‌های مشتق شده می‌شوند.