Table of Contents

Item 23: Prefer non-member non-friend functions to member functions

یک کلاس را در نظر بگیرید که برای کار با یک web browsers استفاده می‌شود. در میان توابع بیشماری که چنین کلاسی باید داشته باشد، برخی از این توابع مانند پاک کردن کش ، پاک کردن تاریخچه مشاهدات، و پاک کردن همه‌ی کوکی‌ها از سیستم است.

```
class WebBrowser {
public:
   void clearCache();
   void clearHistory();
   void removeCookies();
   ....
};
```

بسیاری از کاربران دوست دارند که چنین کاری رو با همدیگه انجام بدهند، بنابراین WebBrowser باید چنین تابعی رو هم پیشنهاد بده:

```
class WebBrowser {
public:
   ....
   void clearEverything();
   ....
};
```

البته که چنین تابعی می‌تواند به صورت یک تابع غیر عضو تعریف شود که توابع مناسب را فراخوانی کند.

```
void clearBrowser(WebBrowser& wb)
{
   wb.clearCache();
   wb.clearHistory();
   wb.removeCookies();
}
```

خب کدام یکی بهتره؟ تابع عضو clearEverything و یا تابع غیر عضو clearBrowser ؟

طبق قواعد شیءگرایی که می‌گوید داده و توابع باید به همدیگر متصل شوند، و این پیشنهاد می‌دهد که تابع عضو انتخاب عاقلانه‌تری است. متاسفانه، چنین پیشنهادی اشتباه می‌باشد. در واقع این پیشنهاد به خاطر درست نفهمیدن معنای شیء گرایی است. قواعد شیءگرایی اشاره دارد که داده‌ها تا جایی که امکان دارد باید کپسوله شوند. و تابع عضو clearEverything در واقع منجر به کپسوله‌سازی کمتری از تابع غیرعضو clearBrowser می‌شود. به علاوه، پیشنهاد تابع غیر عضو باعث افزایش انعطاف‌پذیری برای توابع مربوط به webBrowser می‌شود، و باعث کاهش وابستگی‌ها در زمان کامپایل شده و توسعه‌پذیری webBrowser را افزایش میدهد. بنابراین رویکرد تابع غیر عضو، بهتر از تابع عضو می‌باشد. این مهم است که دلیل آن را بدانیم.

ما با کپسوله‌سازی شروع می‌کنیم. اگر چیزی کپسوله شود، در واقع آن چیز از مشاهده مستقیم مخفی شده است. هر چقدر که چیزی بیشتر کپسوله شود، چیزهای کمتری از آن دیده می‌شود. هر چقدر چیزهای کمتری دیده شود، با انعطاف پذیری بیشتری می‌توانیم آن را تغییر دهیم، چرا که تغییراتی که ما میدهیم به صورت مستقیم توسط کسی دیده نمی‌شود. هر چقدر چیزی را به خوبی کپسوله کرده باشیم، در نتیجه بیشترین توانایی را برای تغییر دادن آن را داریم. و این دلیل آن است که کپسوله‌سازی حایز ارزش است: کپسوله‌سازی به ما انعطاف‌پذیری لازم برای تغییر دادن کد را میدهد و کاربران خیلی محدودی تحت تاثیر این تغییر قرار میگیرند.

فرض کنید که داده همراه با کلاس قرار گرفته است. هر چقدر کد کمتری بتواند این داده را ببیند(یعنی به آن دسترسی داشته باشد)، آن داده بیشتر کپسوله شده است، و ما با آزادی بیشتری می‌توانیم خصوصیات داده‌ای آن شیء را تغییر دهیم، مثل اعضای داده‌ای، نوع شان و غیره. برای یک اندازه‌گیری سرانگشتی برای این که بدانیم چه میزان از کد می‌تواند یک تکه داده را ببینید، می‌توانیم تعداد توابعی که می‌تواند به آن داده دسترسی داشته باشد را بشماریم: هر چقدر توابع بیشتری به آن دسترسی داشته باشد، میزان کپسوله بودن داده کمتر است.

آیتم ۲۲ توضیح داد که داده‌ی عضو باید به صورت private باشد، چرا که اگر این طور نباشند، توابع نامحدودی می‌توانند به این داده‌ها دسترسی داشته باشند و این یعنی هیچگونه کپسوله‌سازی وجود ندارد. برای داده‌های عضوی که خصوصی هستند، تعداد توابعی که دسترسی به آن‌ها دارند، برابر با تعداد توابع عضو کلاس بعلاوه‌ی توابع دوست هستند، از آنجایی که تنها توابع عضو و دوست دسترسی به داده‌های خصوصی دارند. انتخاب بین توابع عضو(که نه تنها به داده‌های خصوصی کلاس دسترسی دارند، بلکه به توابع خصوصی، enum ها و typedefs ها نیز دسترسی دارند) و توابع غیر عضو غیر دوست (که به هیچکدام از این‌ها دسترسی ندارند) که همین کار را برای ما انجام میدهد، منجر به کپسوله‌سازی بیشتری می‌شود، چرا که تعداد توابعی که به داده‌های خصوصی کلاس دسترسی دارند را افزایش

نمیدهد. این نشان میدهد که چرا clearBrowser (تابع غیر-عضو و غیر-دوست) به تابع clearEverything ارجحیت دارد: چرا که منجر به کپسوله‌سازی بیشتری برای کلاس می‌شود.

در این نقطه، دو نکته حایز اهمیت است. اول این که، این دلیل تنها بر روی توابع غیر عضو و غیر دوست قابل اعمال است. توابع دوست دسترسی مشابه با تابع عضو به داده‌های خصوصی کلاس دارند. از نقطه نظر کپسوله‌سازی، انتخاب بین تابع عضو و غیر عضو نیست، بلکه انتخاب بین توابع عضو و توابع غیر عضو غیر دوست می‌باشد.

نکته‌ی دومی که باید به آن اشاره کنیم این است که چون گفتیم که تابع باید غیر عضو کلاس باشد، به این معنی نیست که نمی‌تواند تابع عضو یک کلاس دیگر نباشد. این ممکن است یک اثبات خفیف برای کسانی باشند که از زبان‌هایی استفاده می‌کنند که همه‌ی توابع باید به صورت کلاس باشند(مثل Eiffel ، Java و C#). به طور مثال، ما می‌توانیم clearBrowser را به صورت یک عضو static از یک کلاس ابزاری تعریف کنیم. تا وقتی که عضوی(یا دوستی) از WebBrowser نباشد، تاثیری روی کپسوله‌سازی داده‌های خصوصی WebBrowser نخواهد داشت. در C++ ، یک راه حل طبیعی این است که clearBrowser را به صورت یک تابع غیر عضو با فضای نام WebBrowser تعریف کنیم.

```
namespace WebBrowserStuff {
class WebBrowser {  };
void clearBrowser(WebBrowser& wb)
{  }
}
```

در اینجا ما از یک چیز طبیعی خیلی فراتر رفته‌ایم، البته که namespace برخلاف class ها می‌تواند در چندین سورس فایل پخش شود. این خیلی مهم است، چرا که توابعی مانند clearBrowser به عنوان توابع راحتی شناخته می‌شوند. این که نه تابع عضو هستند و نه تابع دوست، این بدین معنی است که هیچ‌گونه دسترسی خاصی به WebBrowser ندارند، بنابراین نمی‌توانند چیزی را ارایه دهند که مشتری نتواند به تنهایی از یک راه دیگر استفاده کند. به طور مثال، اگر clearBrowser نبود، مشتری می‌تونست به راحتی توابع clearCache و clearHistory و removeCookies را خودش صدا بزند.

یک کلاس مانند WebBrowser ممکن است که توابع راحتی خیلی زیادی داشته باشد، برخی مرتبط با bookmark ها باشد، برخی مرتبط با printing و برخی دیگر برای مدیریت کوکی ها و غیره. به عنوان یک قاعده‌ی کلی، بیشتر مشتری‌ها، علاقه‌مند به تنها برخی از این توابع راحتی هستند. و دلیلی وجود ندارد که یک مشتری تنها به مسایل bookmark علاقه داشته باشد. یک راه حل مستقیم برای این موضوع این است که توابع مرتبط با bookmark را در یک header file جداگانه بنویسیم، و توابع راحتی

cookie-related رو در یک هدر فایل جداگانه قرار دهیم و به همین صورت همه چیز را از هم جدا کنیم:

```
//header "webbrowser.h" -- header for class webBrowser itself
//as well as "core" WebBrowser-releated functionality
namespace WebBrowserStuff {
class WebBrowser {  };
….                              //"core" releated functionality,e.g.
                                //non-member functions almost all clients need
}

//header "webbrowserbookmark.h"
namespace WebBrowserStuff {
                        //bookmark-related convenience functions
}

//header "webbrowsercookies.h"
namespace WebBrowserStuff {
                        //cookie-releated convenience functions
}
```

توجه داشته باشید که این دقیقا راهی است که کتابخانه‌ی استاندارد C++ توسط آن مدیریت می‌شود. به جای آن که یک فایل هدر تنها <Standard library> داشته باشیم، هزاران هدر داریم (vector,algorititthm, memory) داریم که همه در namespace std قرار گرفته‌اند. مشتری‌هایی که تنها نیاز به کاربردهای vectorای دارند نیازی به اضافه کردن هدر memory ندارند، و کاربرانی که نیازی به list ندارند، مجبور نیستند هدر list را اضافه کنند. این به کاربران اجازه میدهد که تنها بخش‌هایی را کامپایل کنند که نیاز دارند.(آیتم ۳۱ را برای بحث در مورد راه‌های دیگری که برای کاهش وابستگی در کامپایل تایم استفاده می‌شود، را ببینید). تقسیم کردن تابع‌ها به این صورت وقتی که تابع عضو باشد امکان‌پذیر نیست، چرا که یک کلاس باید به صورت یکجا تعریف شود و امکان تقسیم کردن آن وجود ندارد.

قرار دادن همه‌ی توابع راحتی در چندین هدر فایل(با یک فضای نام)- به این معنی است که کاربر نیز می‌تواند مجموعه‌ای از توابع راحتی را اضافه کند. همه‌ی چیزی که نیاز دارند انجام بدهند این است که توابع غیر عضو و غیر دوست را به فضای نام اضافه کنند. به طور مثال، اگر یک کاربر WebBrowser بخواهد توابع راحتی‌ای بنویسید که مرتبط با دانلود عکس‌ها باشد، او تنها نیاز خواهد داشت که یک هدر فایل اضافه کند که این توابع در فضای نام WebBrowserStuff تعریف شده باشد. توابع جدید حال در

دستری خواهند بود و همراه با همه‌ی توابع راحتی دیگر شده است. این یک ویژگی دیگری است که کلاس نمی‌تواند چنین چیزی را داشته باشد، چرا که تعاریف کلاس را کاربر نمی‌تواند تغییر بدهد. البته که، کاربران می‌توانند کلاس‌های مشتق شده داشته باشند، ولی کلاس‌های مشتق شده نمی‌توانند به کپسول‌ها دسترسی داشته باشند(یعنی چیزهای خصوصی در کلاس base) . علاوه بر این آیتم ۷ را ببینید، همه‌ی کلاس‌ها طوری طراحی نشده‌اند که بتوانند به عنوان کلاس base استفاده شوند.