

هنر کدنویسی خوانا

تکنیک‌های ساده و کاربردی برای نوشتن کدهای بهتر

The Art of Readable Code

Simple and Practical Techniques for Writing Better Code

شیرین

شیرین‌فوشی

Dustin Boswell
Trevor Foucher

هنر کدنویسی خوانا

The Art of Readable Code

تکنیک‌های ساده و کاربردی برای نوشتن کدهای بهتر

ترجمه:

حسین مسعودی

مشخصات کتاب

سروشناسه	:	مارتین، جان
عنوان و نام پدیدآور	:	هنر کدنویسی خوانا: تکنیک‌های ساده و کاربردی برای نوشتن کدهای بهتر / ترجمه: حسین مسعودی.
شناسه افزوده	:	حسین مسعودی
مشخصات ظاهری	:	۲۳۶ ص. وزیری
شابک	:	۹۷۸-۶۲۲-۶۱۰۳۵۹-۶
وضعیت فهرست‌نویسی	:	فیبا
موضوع	:	برنامه‌نویسی، کدنویسی خوانا
شناسه مجاز	:	۹-۵۸۶۲۰-۳۲۰۲۸۸
انتشارات	:	نگارخانه
پایگاه اینترنتی	:	www.book.tinybit.ir

فهرست مطالب

۱	پیش‌گفتار
۲	این کتاب در مورد چیست؟
۳	شیوه خواندن این کتاب
۵	فصل اول
۵	کد باید به آسانی قابل درک باشد
۶	چه چیزی باعث بهتر شدن کد می‌شود؟
۷	قضیه بنیادی خوانایی
۸	آیا همیشه کوتاه‌تر بودن بهتر است؟
۸	آیا معیار «زمان درک کد» با دیگر اهداف تداخل دارد؟
۹	قسمت سخت فصل
۱۱	بخش اول
۱۱	بهبودهای سطح-ظاهری
۱۳	فصل دوم
۱۳	قرار دادن اطلاعات در نامها
۱۴	انتخاب کلمات خاص
۱۶	پیدا کردن کلمات مشابه
۱۷	اجتناب از نامهای عمومی مانند tmp و retval
۱۸	tmp
۱۹	حلقه تکرار
۲۰	قانون در مورد نامهای عمومی

۲۱	نام‌های واقعی را نسبت به نام‌های انتزاعی ترجیح دهید
۲۱	مثال: DISALLOW_EVIL_CONSTRUCTORS
۲۲	مثال: --run_locally
۲۴	ضمیمه کردن اطلاعات اضافه به یک نام
۲۵	مقدارها و واحدها
۲۶	کد کردن سایر خصوصیات مهم
۲۷	آیا این نماد مجارستانی (Hungarian Notation) است؟
۲۸	طول یک نام چقدر باید باشد؟
۲۸	نام‌های کوتاه برای قلمروهای کوتاه مناسب هستند
۲۹	نوشتن نام‌های طولانی-به هر حال مشکلی ندارد
۳۰	مخفف‌ها و اختصارها
۳۰	دور انداختن کلمات زائد
۳۰	از قالب بندی نام برای انتقال معنی آن استفاده کنید
۳۲	دیگر قراردادهای قالب بندی
۳۳	خلاصه
۳۵	فصل سوم
۳۵	نام‌هایی که نمی‌توانند متناقض باشند
۳۶	مثال: Filter()
۳۷	مثال: Clip(text, length)
۳۸	استفاده از min و max برای (شامل بودن) محدودیت‌ها
۳۹	ارجحیت first و last برای محدوده‌های جامع
۴۰	ارجحیت begin و end برای محدوده‌های جامع/اختصاصی
۴۰	نام‌های Boolean

۴۱	تطابق انتظارات کاربران
۴۱	مثال: <code>get*()</code>
۴۲	<code>list::size()</code> مثال:
۴۳	wizard کیست؟
۴۳	مثال: ارزیابی گزینه‌های مختلف از نامها
۴۶	خلاصه فصل
۴۹	فصل چهارم
۴۹	زیبا سازی
۵۰	زیباسازی در مقایسه با طراحی
۵۱	چرا زیباسازی مهم است؟
۵۲	تنظيم مجدد خطوط شکسته به صورت استوار و جمع و جور
۵۵	از متدها برای پاک کردن بی نظمی استفاده کنید
۵۷	در صورت مفید بودن از ترازبندی ستونی استفاده کنید
۵۸	آیا باید از ترازبندی ستونی استفاده کرد؟
۵۹	یک ترتیب معنا دار انتخاب و از آن به طور مدام استفاده کنید
۵۹	اعلان‌ها را در یک بلوک قرار دهید
۶۱	تقسیم‌بندی کد به صورت پاراگراف‌ها
۶۲	سبک شخصی یا قوانین و قواعد!
۶۴	خلاصه فصل
۶۵	فصل پنجم
۶۵	چه چیزی را کامنت کنیم
۶۷	چه چیزی را نباید کامنت کنیم
۶۸	فقط برای اینکه کدتان کامنت داشته باشد! کامنت گذاری نکنید

برای نامهای بد کامنت ننویسید، در عوض، نامهای بهتری انتخاب کنید	۶۹
ثبت افکار خود	۷۰
افزودن «توضیحات کارگردان».	۷۰
نقصهای موجود در کد خود را کامنت کنید	۷۱
در مورد ثابت‌ها کامنت بنویسید	۷۲
خودتان را جای خواننده بگذارید	۷۳
پیش‌بینی سوالات احتمالی	۷۳
اعلام کردن تله احتمالی	۷۵
کامنت‌های روند کلی	۷۷
کامنت‌های خلاصه	۷۸
دلایل کامنت گذاری: برای چه چیزی؟ چرا؟ یا چگونه؟	۷۹
افکار نهایی، گذر از بلوک نویسنده	۷۹
خلاصه فصل	۸۱
چه چیزی را نباید کامنت کنیم:	۸۱
خود را جای خواننده قرار دهید:	۸۱
فصل ششم	۸۳
ساختن کامنت‌ها به شکل دقیق و خلاصه	۸۳
کامنت‌ها را خلاصه نگه دارید	۸۴
از استفاده از ضمایر مبهم خودداری کنید	۸۴
جملات درهم و برهم لهستانی	۸۵
رفتار یک تابع را به صورت دقیق شرح دهید	۸۵
از مثال‌های ورودی/خروجی که موردهای Corner را نشان می‌دهد استفاده کنید	۸۶
هدف کد خود را مشخص کنید	۸۸

۸۹	کامنت گذاری نام پارامترهای تابع
۹۰	از کلمات Information-Dense استفاده کنید
۹۱	خلاصه فصل
۹۳	بخش دوم
۹۳	ساده سازی حلقه‌های تکرار و منطق
۹۵	فصل هفتم
۹۵	ساده کردن خوانایی کنترل جریان
۹۶	آرگومان‌ها را در شرط‌ها مرتب کنید
۹۸	ترتیب بلوک‌های if/else
۱۰۰	عبارت شرطی (a.k.a. "Ternary Operator) ?:
۱۰۲	از حلقه‌های do/while اجتناب کنید
۱۰۵	دستور بد نام goto
۱۰۶	تو در تو بودن را حداقل کنید
۱۰۷	چگونه تودرتو شدن‌ها روی هم انباسه می‌شود
۱۰۸	حذف تودرتوها با برگرداندن زودتر یک نتیجه
۱۰۸	حذف تودرتوبی داخل حلقه‌ها
۱۱۰	آیا می‌توانید جریان اجرایی برنامه را دنبال کنید؟
۱۱۲	خلاصه فصل
۱۱۳	فصل هشتم
۱۱۳	شکستن عبارت‌های غول پیکر به قسمت‌های کوچک
۱۱۴	متغیرها را شرح دهید
۱۱۵	متغیرهای خلاصه
۱۱۶	استفاده از قوانین دمورگان

سواء استفاده از منطق اتصال کوتاه.....	۱۱۶
مثال: کشمکش با منطق پیچیده.....	۱۱۷
پیدا کردن یک رویکرد ظرفیتر.....	۱۱۹
شکستن دستورات غول پیکر.....	۱۲۱
یکی دیگر از روش‌های خلاقانه برای ساده کردن یک عبارت.....	۱۲۲
خلاصه فصل.....	۱۲۳
فصل نهم.....	۱۲۵
متغیرها و خوانایی.....	۱۲۵
از بین بردن متغیرها.....	۱۲۶
متغیرهای موقتی بی فایده.....	۱۲۶
از بین بردن نتایج واسط.....	۱۲۷
از بین بردن متغیرهای کنترل جریان.....	۱۲۸
دامنه متغیرهای خود را کوچک کنید.....	۱۲۹
محدوده دستور IF در زبان C++.....	۱۳۱
ایجاد متغیرهای در JavaScript Private.....	۱۳۲
دامنه سراسری JavaScript.....	۱۳۳
نبود محدوده‌ها در زبان Python و JavaScript.....	۱۳۴
ترجیح دادن متغیرهای Write-Once.....	۱۳۷
مثال پایانی.....	۱۳۸
خلاصه فصل.....	۱۴۲
بخش سوم.....	۱۴۳
سازماندهی مجدد کد.....	۱۴۳
فصل دهم.....	۱۴۵

۱۴۵	استخراج زیرمسئله‌های غیر مرتبط
۱۴۶	مثال مقدماتی: <code>findClosestLocation()</code>
۱۴۸	کدهایی با کاربرد خاص
۱۴۹	سایر کدهای همه منظوره
۱۵۰	مزایای غیرمنتظره
۱۵۲	ساختن کدهای همه منظوره به تعداد زیاد
۱۵۲	آیا این برنامه‌نویسی بالا-به-پایین است یا پایین-به-بالا؟
۱۵۳	قابلیت‌های پروژه-محور
۱۵۴	ساده سازی <code>Interface</code> موجود
۱۵۶	تغییر شکل مجدد <code>Interface</code> با توجه به نیاز
۱۵۷	دور نگه داشتن بیش از حد توابع از یکدیگر
۱۵۸	خلاصه فصل
۱۵۸	پیشنهاد مطالعه بیشتر
۱۶۱	فصل یازدهم
۱۶۱	در هر لحظه یک وظیفه
۱۶۳	وظیفه‌ها می‌توانند کوچک باشند
۱۶۵	استخراج مقدارها از یک <code>Object</code>
۱۶۷	اعمال کردن «یک وظیفه در یک لحظه»
۱۶۹	یک رویکرد دیگر
۱۷۰	یک مثال بزرگتر
۱۷۲	بهبودهای بیشتر
۱۷۳	خلاصه فصل
۱۷۵	فصل دوازدهم

۱۷۵	تبديل افکار به کد
۱۷۶	توضیح منطق کد به طور شفاف
۱۷۸	اطلاع از کتابخانه‌های تان می‌تواند مفید باشد
۱۷۹	اعمال این رویکرد در مسئله‌های بزرگتر
۱۸۱	یک توضیح به زبان ساده از راه حل
۱۸۲	اعمال متدها به صورت بازگشتی
۱۸۵	خلاصه فصل
۱۸۷	فصل سیزدهم
۱۸۷	نوشتن کد کمتر
۱۸۸	برای پیاده سازی برخی ویژگی‌ها خود را به زحمت نیندازید، به آن نیازی ندارید
۱۸۸	شکستن نیازمندی‌ها با مطرح کردن سوالات مختلف
۱۸۸	مثال: فروشگاه یاب
۱۸۹	مثال: افزودن حافظه نهان یا Cache
۱۹۱	کدپایه خود را کوچک نگه دارید
۱۹۳	با کتابخانه‌های موجود در اطراف خود آشنا باشید
۱۹۳	مثال: لیست‌ها و مجموعه‌ها در Python
۱۹۴	چرا استفاده مجدد از کتابخانه‌ها چنین موفقیتی دارد؟
۱۹۴	مثال: استفاده از ابزارهای Unix به جای نوشتن کد
۱۹۶	خلاصه فصل
۱۹۷	بخش چهارم
۱۹۷	موضوعات برگزیده
۱۹۹	فصل چهاردهم
۱۹۹	انجام تست و خوانایی

۲۰۰	خوانایی تست‌های خود را ساده و قابل نگه‌داری کنید
۲۰۱	مشکل این تست چیست؟!
۲۰۲	تصحیح این تست به شکل خواناتر
۲۰۳	ساختن حداقل دستورات تست
۲۰۴	پیاده سازی سفارشی «Minilanguages»
۲۰۶	پیام‌های خطرا را به شکل خوانا و صحیح بنویسید
۲۰۶	استفاده از نسخه بهتری از <code>assert()</code>
۲۰۸	پیام خطاهای دست ساز
۲۰۸	انتخاب ورودی‌های خوب برای تست
۲۰۹	ساده سازی مقادیر ورودی
۲۱۱	تست‌های چندگانه عملکرد
۲۱۱	نام‌گذاری توابع تست
۲۱۳	مشکل این تست چیست؟
۲۱۴	توسعه <code>Test-Friendly</code>
۲۱۷	جزئیات بیشتر
۲۱۸	خلاصه فصل
۲۱۹	فصل پانزدهم
۲۱۹	طراحی و پیاده سازی یک «شمارنده دقیقه/ساعت»
۲۲۰	مسئله
۲۲۰	تعريف <code>Interface</code> کلاس
۲۲۱	بهبود نام‌ها
۲۲۲	بهبود کامنت‌ها
۲۲۴	تلash شماره ۱: یک راه حل ساده و بی تکلف

۲۲۵	آیا در ک این کد آسان است؟
۲۲۶	نسخه ساده‌تر از نظر خوانایی
۲۲۷	مشکلات کارایی
۲۲۸	تلاش شماره ۲: طراحی تسمه نقاله
۲۲۹	پیاده سازی طراحی تسمه نقاله دو مرحله‌ای
۲۳۰	آیا کار ما تمام شده است؟
۲۳۱	تلاش شماره ۳: طراحی یک Time-Bucketed
۲۳۲	پیاده سازی طراحی Time-Bucketed
۲۳۴	پیاده سازی TrailingBucketCounter
۲۳۶	پیاده سازی ConveyorQueue
۲۳۷	مقایسه سه راه حل
۲۳۸	خلاصه فصل
۲۳۹	Appendix
۲۳۹	پیشنهادهایی برای مطالعه
۲۴۰	کتابهایی برای نوشتن کد با کیفیت بالا
۲۴۱	کتابهایی برای زبان‌های مختلف برنامه‌نویسی
۲۴۲	کتابهایی درباره نکات تاریخی

من بدون عادت به وقت‌شناسی، نظم، تلاش و بدون اراده برای تمرکز بر روی یک موضوع در هر زمان، هرگز نمی‌توانستم به موفقیت‌هایی که بدست آورده‌ام برسم. چارلز دیکنژ.

سخن مترجم

کتاب پیش‌رو ترجمه کتاب *The Art of Readable Code* (برنامه نویسی) اثر Dustin Boswell مهندس نرم افزار شرکت گوگل در سال‌های ۲۰۰۴ تا ۲۰۰۹ و Trevor Foucher است که حاصل سال‌ها تجربه در زمینه برنامه نویسی و همچنین نتایج بررسی هزاران کد مختلف در پروژه‌های گوناگون را در این کتاب گرد آورده است.

خوانندگان عزیز توجه داشته باشند که در راستای فهم و درک بهتر مطالب، برخی مثال‌ها به گونه‌ای ترجمه شده است، تا کاربران فارسی زبان راحت‌تر بتوانند مطلب را درک کنند.

با توجه به دقت فراوانی که در جهت ارائه ترجمه مناسب و قابل فهم برای شما خوانندگان عزیز صورت گرفته است، ممکن است اشکالاتی هر چند کوچک نیز وجود داشته باشد. لذا از تمام عزیزان خواهشمندم که پیشنهادها یا انتقادات خود جهت اصلاح یا بهتر شدن این کتاب، از طریق سایت book.tinybit.ir ارسال نمایند. در صورتی که این کتاب بصورت غیرمجاز بدست شما رسیده است خواشمند است جهت تهیه قانونی آن به آدرس book.tinybit.ir مراجعه نمایید.

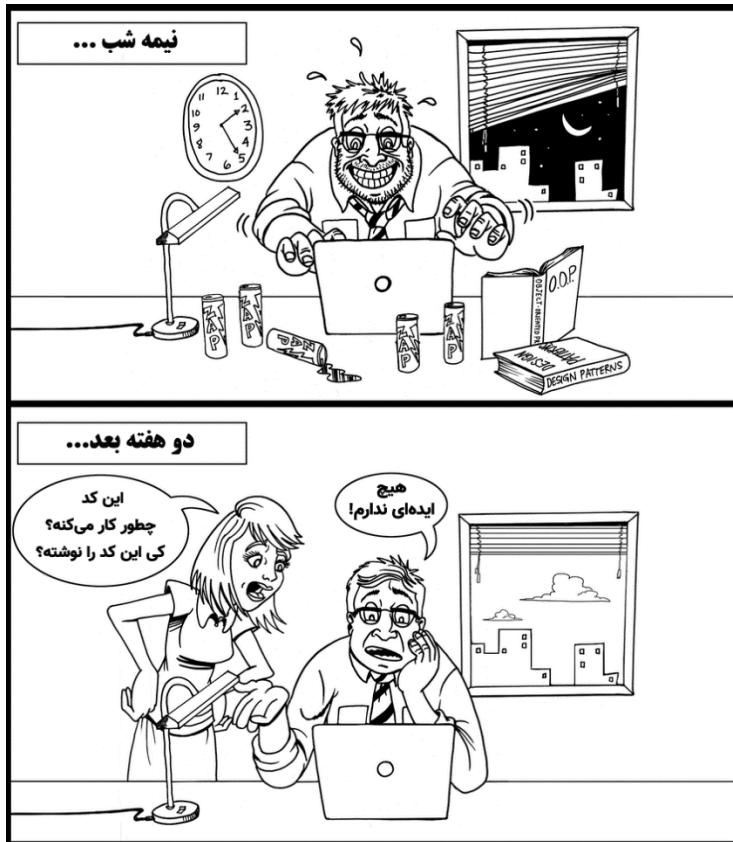
همچنین از همراهان گرامی به خاطر رحمات فراوانی که در تصحیح و ویرایش نهایی کتاب کشیده‌اند، تشکر و قدردانی می‌نمایم. امید آنکه در سایه الطاف خداوند متعال همواره سلامت و موفق باشند.

در پایان نیز به برنامه‌نویسان عزیز پیشنهاد می‌کنم در صورت تمایل حتماً کتاب اثر مرکب اثر دارن هاردی را در اولین فرصت مطالعه نمایید، چرا که باعث می‌شود بتوانید یک برنامه مدون و اصولی را بدور از ایده‌آل گرایی‌های دست نیافتی، کنار گذاشته و با تصمیمات کوچک و موثر در بازه‌های زمانی بلند مدت، با تمرین و ممارست به اهداف خود برسید.

حسین مسعودی

Hossein52hz@gmail.com

پیش‌گفتار



با اینکه ما در شرکت‌های نرم افزاری بسیار موفق، با مهندسین برجسته‌ای کار کرده‌ایم، بسیار اتفاق افتاده است که با کدی روبرو شده‌ایم که هنوز هم فضای زیادی برای بهبود دارد. در واقع، کدهایی را دیده‌ایم که برخی از آن‌ها واقعاً رشت بوده‌اند و شما نیز احتمالاً چنین کدهایی را دیده‌اید. اما وقتی کدی می‌بینیم که به زیبایی نوشته شده است، این می‌تواند الهام بخش‌ا و انگیزه‌ای برای نوشتمن کدهای خوب باشد. یک کد خوب می‌تواند بسیار سریع به شما آموزش دهد که چه چیزی در حال انجام است، استفاده از آن جالب بوده و به شما انگیزه‌می‌دهد که کدهای خودتان را بهتر کنید. هدف این کتاب کمک کردن به شما برای بهتر کردن کدهایتان است. وقتی می‌گوییم «کد»، منظور خطوطی از کد است که شما در ویرایشگر خود شروع به نوشتمن می‌کنید. ما درباره معماری پروژه شما یا انتخاب الگوی طراحی شما صحبتی نمی‌کنیم. قطعاً این موارد نیز مهم هستند ولی آنچه به تجربه برای ما ثابت شده است این است که زندگی روزمره یک شخص به عنوان برنامه‌نویس، بیشتر برای چیزهای پایه مانند نام‌گذاری متغیرها، نوشتمن حلقه‌ها و حمله به مشکلات و شکستن آن‌ها در سطح تابع، صرف می‌شود که بخش عمده‌ای از آن خواندن و ویرایش کدی است که در حال حاضر وجود دارد. امیدواریم این کتاب را برای برنامه‌نویسی روزانه خود بسیار مفید یافته و خواندن آن را به دیگر برنامه‌نویسان نیز پیشنهاد دهید.

این کتاب در مورد چیست؟

موضوع این کتاب در مورد شیوه نوشتمن کدی است که حداقل خوانایی را داشته باشد و ایده اصلی آن این است که کد باید به آسانی قابل درک باشد. خصوصاً که باید هدف شما به حداقل رساندن مدت زمان لازم، برای درک کد نوشتمن شده باشد. در این کتاب این ایده را شرح داده و آن را با مثال‌های زیادی از زبان‌های مختلف از جمله، C++, Python و JavaScript توضیح می‌دهیم. البته اگر با همه این زبان‌ها آشنا نیستید، می‌توانید به راحتی مثال‌ها را دنبال کنید چرا که در این کتاب از پرداختن به ویژگی‌های پیشرفتمن زبان‌های برنامه‌نویسی اجتناب کرده‌ایم، (در تجربه ما، مفاهیم خوانایی یک کد عمدتاً مستقل از نوع زبان هستند). هر فصل از این کتاب به جنبه متفاوتی از کدنویسی و اینکه چطور یک کد را برای درک ساده‌تر، تغییر دهید، می‌پردازد. کتاب به چهار بخش تقسیم شده است:

بخش بهبودهای سطح‌ظاهری^۱:

این بخش شامل نام‌گذاری، کامنت‌گذاری و زیباسازی^۲ یک کد است که این‌ها ترفندهای ساده‌ای هستند که در هر خط از کدپایه^۳ شما اعمال می‌شوند.

ساده‌سازی حلقه‌ها و منطق کد:

این بخش شامل روش‌هایی برای پالایش حلقه‌ها، متغیرها و منطق در برنامه شما است، تا درک آن‌ها ساده‌تر شود.

سازماندهی مجدد کد:

این بخش، روش‌های اعمال شده در سطح‌بالاتر که برای سازماندهی بلوک‌های بزرگ کد و حمله^۴ به مسئله‌ها و شکستن آن‌ها در سطح تابع است را شامل می‌شود.

موضوعات برگزیده:

این بخش شامل اجرای راهکارهای «درک راحت‌تر» در عملیات تست نویسی و ارائه یک مثال کدنویسی با ساختار داده بزرگ است، که بخش پایانی کتاب خواهد بود.

شیوه خواندن این کتاب

این کتاب در نظر دارد که خواندن آن سرگرم کننده و گاه به گاه باشد. امیدواریم، خوانندگان این کتاب را در یک یا دو هفته مطالعه کنند. فصل‌ها به ترتیب دشورای نوشته شده‌اند: مباحث ساده در ابتدا آورده شده و مباحث پیشرفته‌تر در انتهای قرار دارند. با این حال، هر فصل به صورت مستقل است و می‌تواند به صورت جداگانه مورد مطالعه قرار گیرد. همچنین در صورت تمایل می‌توانید فصلی را نادیده گرفته و به سراغ فصل دیگر بروید.

^۱ Surface-level

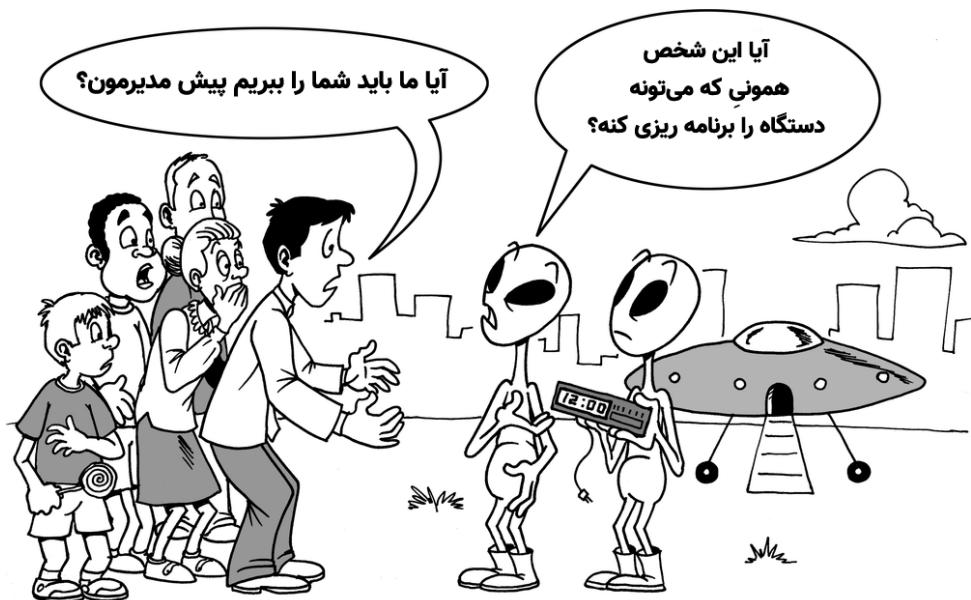
^۲ aesthetics

^۳ codebase

^۴ attack

فصل اول

کد باید به آسانی قابل درک باشد



حدود پنج سال قبل، ما هزاران مثال از کدهای بد را جمع آوری کردیم (که البته اکثرشان متعلق به خود ما بود)، سپس آن‌ها را از این جهت که چه عاملی سبب بد شدن آن‌ها شده بود و نیز قوانین/تکنیک‌هایی که برای بهتر شدن آن‌ها استفاده کرده بودیم را مورد بررسی قرار دادیم. آنچه متوجه شدیم این بود که همه اصول ناشی از یک مورد واحد است.

**کلید طلایی
کد باید به آسانی درک شود.**

ما معتقدیم این کلید طلایی، مهمترین اصلی است که می‌توانید هنگام تصمیم گیری در مورد نوشتن کد خود استفاده کنید. در طول این کتاب، نشان خواهیم داد که چگونه می‌توانید این اصل را در جنبه‌های مختلف کدنویسی‌های روزانه خود بکار ببرید. اما قبل از شروع، اجازه دهید در این مورد که چرا این اصل اینقدر مهم است، توضیح دهیم.

چه چیزی باعث بهتر شدن کد می‌شود؟

اکثر برنامه‌نویسان (از جمله خود من) تصمیمات برنامه‌نویسی را بر اساس احساسات و برداشت ناگهانی می‌گیرند.

همه ما می‌دانیم که کدی شبیه این:

```
for (Node* node = list->head; node != NULL; node = node->next)
    Print(node->data);
```

بهتر از کدی شبیه به این یکی است:

```
Node* node = list->head;
if (node == NULL) return;
while (node->next != NULL) {
    Print(node->data);
    node = node->next;
}
if (node != NULL) Print(node->data);
```

حتی اگر فکر کنید که هر دو مثال دقیقاً شبیه هم عمل می‌کنند، اما در بسیاری از مواقع این انتخاب سخت است، برای مثال کد زیر را ببینید:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

نسخه اول فشرده‌تر است، اما نسخه دوم کمتر ترسناک است. به نظر شما کدام معیار اهمیت بیشتری دارد؟ فشردگی یا ترسناک نبودن؟ به طور کلی چگونه تصمیم می‌گیرید که با چه معیاری کدی را بنویسید؟

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

قضیه بنیادی خوانایی

بعد از مطالعه کدهای شبیه کدهای بالا، ما به این نتیجه رسیدیم که یک معیار برای خوانایی وجود دارد که مهم‌تر از بقیه موارد است. این معیار خیلی مهم است به همین دلیل آن را قضیه بنیادی خوانایی نام‌گذاری می‌کنیم.

کلید طلایی

کد باید به گونه‌ای نوشته شود که زمان فهمیدن آن توسط شخص دیگر، کمترین میزان ممکن باشد.

منظور ما از این جمله چیست؟ به معنی واقعی کلمه اگر شما از یک همکار معمولی خود بخواهید که کدتان را بخواند و مقدار زمانی که او کد شما را خوانده و درک می‌کند را اندازه بگیرید، این «زمان تا فهمیدن» معیاری تئوری خواهد بود که شما برای حداقل بودن زمان درک کدتان نیاز دارید.

زمانی که می‌گوییم «درک کردن»، واژه درک کردن، بار معنایی زیادی دارد. برای اینکه کسی کد شما را به طور کامل درک کند، باید بتواند در آن تغییرات ایجاد کرده، اشکالات آن را پیدا کند و نحوه تعامل این کد با بقیه برنامه شما را بفهمد.

ممکن است به این فکر کنید که چه کسی اهمیت می‌دهد که شخص دیگری بتواند کد را درک کند یا نه؟ من تنها کسی هستم که از این کد استفاده می‌کنم! ولی بدانید که حتی اگر در یک پروژه تک نفری هستید، دنبال کردن این هدف ارزشمند است، چراکه ممکن است شش ماه بعد، یعنی زمانی که کدتان برای شما نآشنا شده، آن شخص دیگر، خود شما باشید، همچنین ممکن است شخص دیگری به پروژه شما بپیوندد یا کد شما در پروژه دیگری استفاده شود، پس هم اکنون ارزش دارد که کد خود را قابل درک بنویسید.

آیا همیشه کوتاهتر بودن بهتر است؟

به طور کلی، هرچه کد کمتری برای حل یک مسئله بنویسید، بهتر است.(به فصل ۱۳ مراجعه کنید). بیشک درک یک کلاس ۲۰۰۰ خطی نسبت به یک کلاس ۵۰۰۰ خطی زمان کمتری از شما می‌گیرد.

اما خطوط کمتر همیشه هم بهتر نیست! در موقع زیادی شما با یک عبارت تک خطی، شبیه این عبارت مواجه می‌شوید:

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

این حالت، نسبت به حالتی که عبارت دو خطی است، زمان بیشتری برای درک کد از شما می‌گیرد:

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

به طور مشابه، یک کامنت می‌تواند سبب درک سریعتر کد شود، حتی اگر کد بیشتری را به فایل اضافه کند:

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```

بنابراین هر چند داشتن خطوط کد کمتر هدف خوبی است، ولی به حداقل رساندن زمان درک^۱ کد، هدفی بهتر است.

آیا معیار «زمان درک کد» با دیگر اهداف تداخل دارد؟

احتمالاً به این فکر می‌کنید که، پس سایر محدودیتها را چه کنیم؟ مثلًاً کارآمدی^۲ بهتر کد، یا معماری بهتر یا ساده بودن کد برای تست، یا دیگر موارد؟ آیا گاهی اوقات، این موارد با کاری که می‌خواهید برای راحتتر درک شدن کد انجام دهید، تداخل ندارند؟

^۱ Time-Till-Understanding

^۲ efficient

ما متوجه شدیم هیچ تداخلی با اهداف دیگر وجود ندارد. حتی در حوزه کدهای بسیار بهینه، هنوز می‌شود با روش‌هایی آن‌ها را بسیار خواناتر نمود. نوشتمن کد به شکلی که ساده‌تر درک شود، در موارد زیادی کدهایی را نتیجه می‌دهد که معماری خوبی داشته و تست آن‌ها نیز ساده‌تر است.

در ادامه کتاب در مورد چگونگی «ساده خوانده شدن کد» در شرایط مختلف بحث خواهد شد. اما هر زمان که چهار تردید شدید، به یاد داشته باشید که «قضیه اساسی خوانایی»، هر قانون یا اصل دیگری در این کتاب را به چالش کشیده و همیشه حرف اول را می‌زند.

همچنین برخی از برنامه‌نویس‌ها برای تصحیح کدهایی که به خوبی بازسازی نشده‌اند نیاز به اجبار دارند. همیشه این سوال مهم است که برگردیم و بپرسیم، آیا این کد برای درک شدن آسان است؟ اگر بله، احتمالاً خوب است، پس بررسی کدهای دیگر را ادامه می‌دهیم.

قسمت سخت فصل

بله، شما به تمرین‌های زیادی نیاز دارید. این تمرین که باید به طور دائم شخصی خیالی را که بتواند به راحتی کد شما را درک کند، در نظر بگیرید. انجام این کار مستلزم روشن کردن بخشی از مغز است که ممکن است قبلًا هنگام کدنویسی خاموش بوده باشد.

اگر این هدف را در نظر داشته باشید (همانگونه که هدف ما است)، یقین داریم که به یک کدنویس بهتر تبدیل خواهید شده و کمتر با گ خواهید داشت. همچنین بیشتر به کار خود افتخار نموده و کدی تولید خواهید کرد که هر شخص دیگری مشتاقانه از آن استفاده خواهد نمود. پس باید شروع کنیم!

بخش اول

بهبودهای سطح-ظاهری^۱

ما هدف خوانایی را با در نظر گرفتن بهبودهای «سطح-ظاهری» به این صورت شروع می‌کنیم: انتخاب نامهای خوب، نوشتن کامنت‌های مفید و قالب‌بندی مرتب کدهای خود.

این نوع تغییرات به راحتی قابل اعمال بوده و می‌توانید بدون نیاز به بازسازی^۲ کد یا تغییر چگونگی اجرای برنامه، آنها را در جای خود ایجاد کنید. این امر می‌تواند به صورت تدریجی و بدون سرمایه گذاری زمان زیادی، صورت بگیرد.

از آنجایی که این موارد روی هر خط از کد شما در کدپایه اثر می‌گذارند، بسیار مهم هستند. اگرچه هر تغییر ممکن است به تنها یک کوچک بنظر برسد، ولی در مجموع می‌تواند پیشرفت چشم‌گیری از در کدپایه شما ایجاد کند. اگر کد شما دارای نامهای عالی، کامنت‌های مناسب و استفاده مفید از فضای خالی باشد، کد شما خوانایی بسیار بیشتری خواهد داشت. البته موارد بسیار دیگری نیز هستند که لایه‌هایی پایین‌تر از این سطح-ظاهری قرار دارند و خوانایی کد را بیشتر می‌کنند که در قسمت‌های بعدی کتاب آنها را بررسی خواهیم کرد.

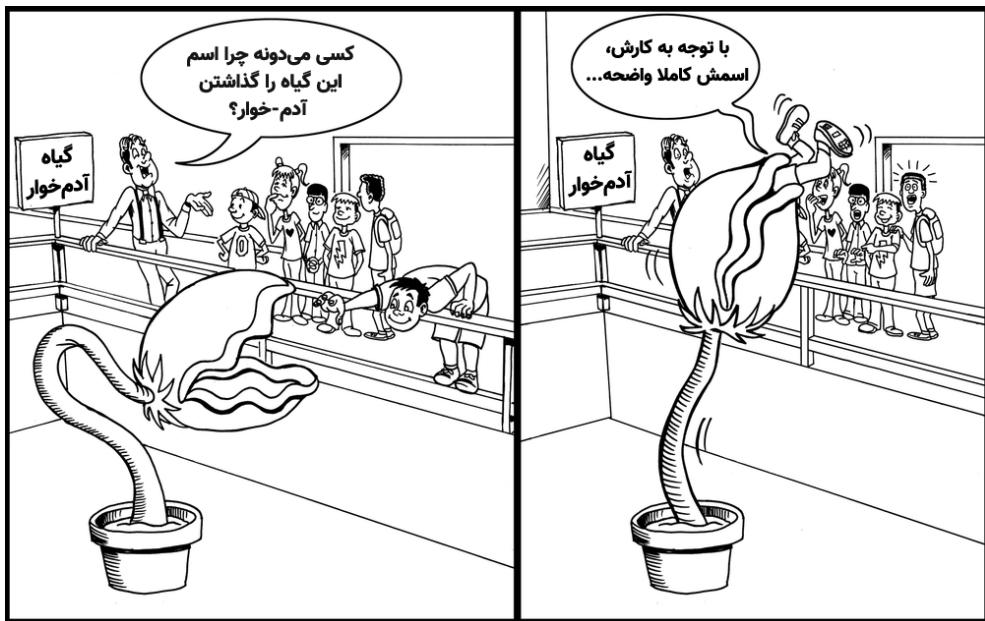
مطلوب این بخش بسیار کاربردی است که این ارزش را دارد که در ابتدا در مورد این اثر گذاری‌های کوچک صحبت کنیم.

^۱ Surface-Level

^۲ refactor

فصل دوم

قرار دادن اطلاعات در نام‌ها^۱



^۱ Packing information in names

در نام گذاری یک متغیر، یک تابع یا یک کلاس، تعداد زیادی اصول مشابه وجود دارد که باید در آن‌ها اعمال کنید. ما دوست داریم به یک نام، به عنوان یک کامنت کوچک فکر کنیم. حتی اگر فضای زیادی وجود ندارد، می‌توانید اطلاعات زیادی را با انتخاب یک نام خوب به خواننده منتقل کنید.

کلید طلایی
اطلاعات را در نام‌های خود قرار دهید.

بسیاری از نام‌هایی که ما در برنامه‌نویسی می‌بینیم مبهم هستند، مانند tmp. حتی کلماتی که ممکن است منطقی به نظر برسند مانند size یا get، نیز اطلاعات زیادی را در خود جای نمی‌دهند. این فصل به شما نشان می‌دهد چگونه نام‌هایی انتخاب کنید که اطلاعات زیادی را در خود دارند.

این فصل به شش موضوع خاص می‌پردازد:

- انتخاب کلمات خاص
- اجتناب از نام‌های عمومی (یا دانستن اینکه چه زمانی از آن‌ها استفاده کنید)
- استفاده از نام‌های واقعی^۱ بجای نام‌های انتزاعی
- افزودن اطلاعات اضافی به یک نام با استفاده از پیشوند یا پسوند
- تصمیم گیری در مورد اینکه یک نام چه مدت باید استفاده شود
- استفاده از قالب بندی نام برای افزودن اطلاعات اضافی

انتخاب کلمات خاص

بخشی از قرار دادن اطلاعات در نام‌ها به انتخاب کلماتی بازمی‌گردد که بسیار خاص هستند. در این بین باید به اجتناب از کلمات تهی^۲ دقت داشته باشید.

به عنوان مثال کلمه get در مثال زیر خیلی غیرخاص است:

```
def GetPage(url):
    ...
```

^۱ concrete

^۲ empty

کلمه get چیز زیادی به ما نمی‌گوید. آیا این متدهای get یک صفحه را از یک cache محلی می‌گیرد یا از یک دیتابیس و یا از اینترنت؟ اگر منظور از get در اینجا این است که چیزی را از اینترنت می‌گیرد یک کلمه خاص‌تر همچون FetchPage() یا DownloadPage() گزینه بهتری است.

در اینجا مثالی از یک کلاس BinaryTree یا درخت جستجوی دودویی را داریم:

```
class BinaryTree {
    int Size();
    ...
};
```

شما انتظار دارید که متدهای size() چه چیزی را به عنوان خروجی برگرداند؟ ارتفاع درخت؟ تعداد گره‌ها؟ یا میزان مصرف حافظه از لحظه پیمایش درخت؟

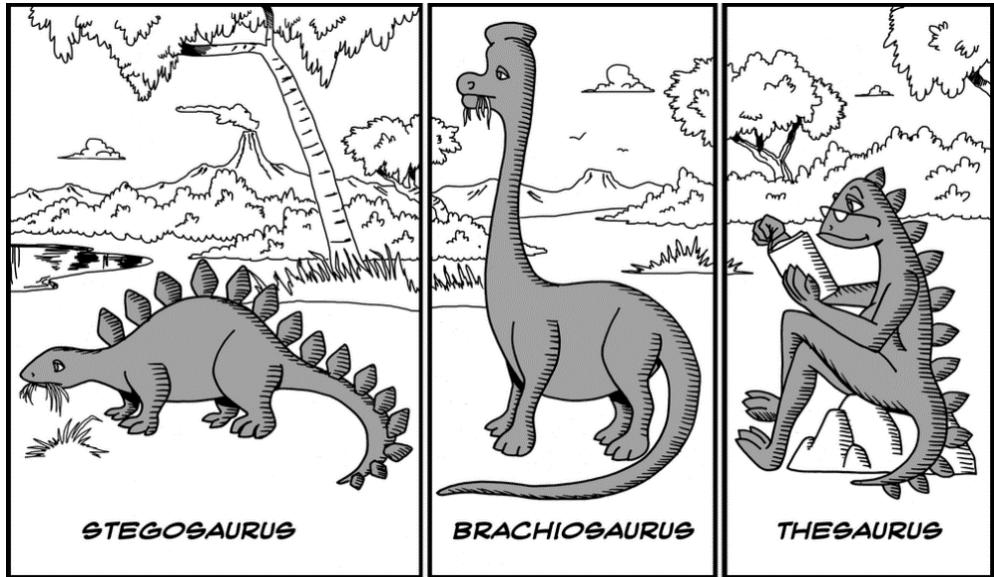
مشکل این است که size() اطلاعات زیادی را به خواننده انتقال نمی‌دهد. می‌توان از یک نام کمی خاص‌تر همچون NumNodes() یا Height() استفاده کرد.

به عنوان مثالی دیگر، فرض کنید یک کلاس مرتب سازی از Thread دارید:

```
class Thread {
    void Stop();
    ...
};
```

نام Stop() مناسب است اما بستگی به این دارد که دقیقاً چه کاری انجام می‌دهد. احتمالاً نام خاص‌تر نیز وجود داشته باشد، برای نمونه اگر یک عملیات سنگین برگشت ناپذیر است احتمالاً استفاده از Kill() به جای آن بهتر خواهد بود. یا اگر راهی برای Resume() کردن آن وجود داشته باشد، بهتر است از Pause() استفاده کنید.

پیدا کردن کلمات مشابه



از اینکه از یک فرهنگ لغت استفاده کنید یا از دوست خود درخواست کنید که نام بهتری را پیشنهاد دهد، نترسید. زبان انگلیسی یک زبان غنی است و کلمات بسیار زیادی برای انتخاب شما وجود دارند.

در اینجا مثال‌های مختلفی را برای یک کلمه آورده‌ایم، نسخه‌های مشابه مختلفی که ممکن است با توجه به شرایط شما، کاربرد داشته باشند:

معادل/متراffدها	کلمه
deliver, dispatch, announce, distribute, route	Send
search, extract, locate, recover	Find
launch, create, begin, open	Start
create, set up, build, generate, compose, add, new	Make

با این حال افراط نکنید. در زبان PHP تابعی برای `explode()` یک رشته وجود دارد. این اسمی است که می‌توان معانی مختلفی از آن برداشت کرد و تصویر خوبی از شکستن چیزی به تکه‌های مختلف را نشان می‌دهد، اما تفاوت آن با `(split()` چیست؟ (این‌ها دو تابع مختلف هستند، اما حدس زدن تفاوت آن‌ها بر اساس نام‌شان کار سختی است).

کلید طلایی

بهتر است نام گذاری شفاف و دقیق باشد تا اینکه باحال به نظر برسد.

اجتناب از نام‌های عمومی مانند tmp و retval

نام‌هایی مانند tmp، foo و retval معمولاً نشان دهنده بهانه هستند، به این معنی که من نمی‌توانم به یک نام خوب فکر کنم. بجای استفاده از یک نام تهی مانند این، نامی انتخاب کنید که ارزش موجودیت یا هدف شما را توصیف کند.

به عنوان مثال در تابع javascript زیر از retval استفاده شده است:

```
var euclidean_norm = function (v) {
  var retval = 0.0;
  for (var i = 0; i < v.length; i += 1)
    retval += v[i] * v[i];
  return Math.sqrt(retval);
};
```

استفاده از retval زمانی که نمی‌توانید برای برگرداندن مقدار، به یک نام بهتر فکر کنید، و سوشه انگیز است. اما retval اطلاعات بیشتری نسبت به این جمله که «من یک مقدار برگشت داده شده هستم» ندارد.

یک نام بهتر هدف متغیر یا ارزش موجود در آن را توصیف می‌کند. در این مورد، متغیر مجموع توان دوم‌های v را حساب می‌کند. پس sum_squares نام بهتری است. این نام هدف متغیر را که از کجا آمده آشکار می‌کند و ممکن است در رفع یک اشکال کمک کند.

به عنوان نمونه، تصور کنید که این متغیر داخل یک حلقه به صورت تصادفی بوده است:

```
retval += v[i];
```

اگر نام متغیر sum_squares باشد خیلی واضح‌تر است:

```
sum_squares += v[i]; // Where's the "square" that we're summing? Bug!
```

توصیه

نام retval اطلاعات زیادی را داخل خود ندارد. به جای آن از یک نام که مقدار متغیر را توصیف می‌کند استفاده کنید.

با این وجود در برخی موارد، نام‌های عمومی معنی کاری که انجام می‌دهند را شامل می‌شوند. بباید به زمانی که استفاده از آن‌ها منطقی به نظر می‌رسد نگاهی بیندازیم.

tmp

یک مورد کلاسیک از جایجایی دو متغیر را در نظر بگیرید:

```
if (right < left) {
    tmp = right;
    right = left;
    left = tmp;
}
```

در مواردی مشابه کد بالا، نام tmp کاملا خوب است. تنها هدف متغیر، ذخیره سازی موقت با عمری کوتاه در چند خط است. نام tmp معنی خاصی را به خواننده منتقل می‌کند به این معنی که این متغیر هیچ وظیفه دیگری ندارد. این متغیر به تابع دیگری ارسال نمی‌شود و یا ریست یا مورد استفاده مجدد قرار نمی‌گیرد.

اما در مثال بعدی، tmp بیهوده استفاده شده است(کافی بود به جای tmp موارد جدید را به همان اضافه می‌کردیم و نیازی به tmp نبود):

```
String tmp = user.name();
tmp += " " + user.phone_number();
tmp += " " + user.email();
...
template.set("user_info", tmp);
```

حتی اگر این متغیر دارای طول عمر کوتاهی باشد، ذخیره سازی موقت، مهمترین چیز درباره این متغیر نیست. در عوض، نامی مثل user_info توصیف بیشتری را دارد.

در مثال زیر tmp باید در نام باشد، اما فقط به عنوان قسمتی از آن:

```
tmp_file = tempfile.NamedTemporaryFile()
...
SaveData(tmp_file, ...)
```

توجه داشته باشید که ما نام متغیر را tmp_file نام گذاری کردیم و نه فقط tmp، چون یک شیء فایل است. تصور کنید اگر ما فقط آن را tmp صدا می‌زدیم چه اتفاقی می‌افتد:

```
SaveData(tmp, ...)
```

تنها با نگاه به همین یک خط از کد، مشخص نمی‌شود که tmp یک فایل است یا اسم یک فایل و یا حتی ممکن است داده‌ای باشد که در جایی نوشته خواهد شد.

توصیه

نام tmp باید تنها در مواردی استفاده شود که متغیر عمر کوتاهی داشته و موقعی بودن آن، مهمترین حقیقت درباره آن باشد.

حلقه تکرار

نام‌هایی مانند i، j، و it معمولاً به عنوان شاخص و تعداد تکرار حلقه استفاده می‌شوند. اگرچه این اسامی عمومی هستند، اما به معنی «من یک شمارنده هستم» شناخته می‌شوند. در حقیقت، اگر یکی از این نام‌ها را برای اهداف دیگری استفاده کنید باعث سردرگمی دیگران می‌شود، پس این کار را نکنید.

اما بعضی اوقات نسبت به i و j و k نام‌های بهتری نیز وجود دارد. برای نمونه در حلقه‌های زیر به دنبال این هستیم که بفهمیم کدام کاربر متعلق به کدام club است:

```
for (int i = 0; i < clubs.size(); i++)
    for (int j = 0; j < clubs[i].members.size(); j++)
        for (int k = 0; k < users.size(); k++)
            if (clubs[i].members[k] == users[j])
                cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

در دستور if موارد members[] و users[] از index اشتباهی استفاده می‌کنند. اشکالاتی شبیه این به سختی کشف می‌شوند، چرا که این خط از کد در حالت جداگانه^۱ خوب به نظر می‌رسد.

```
if (clubs[i].members[k] == users[j])
```

در این مورد استفاده از نام‌های دقیق‌تر کمک بیشتری می‌کند. به جای استفاده از شمارنده‌های حلقه (i، j، k)، انتخاب مناسب‌تر (club_i، members_i، users_i) است که به شکل خلاصه‌تر می‌توان از (ci، mi، ui) استفاده نمود:

```
if (clubs[ci].members[ui] == users[mi]) # Bug! First letters don't match up.
```

در صورت استفاده صحیح، حرف اول شمارنده با حرف اول آرایه مطابقت خواهد داشت:

^۱ isolation

```
if (clubs[ci].members[mi] == users[ui]) # OK. First letters match.
```

قانون در مورد نام‌های عمومی

همان گونه که متوجه شدید، در شرایطی نام‌های عمومی مفیدتر هستند.

توصیه

اگر می‌خواهید از یک نام عمومی مانند tmp، it یا retval استفاده کنید، حتماً دلیل خوبی برای انجام این کار داشته باشید.

این قابل درک است که وقتی هیچ کلمه بهتری به ذهنتان نرسید، راحت‌ترین کار به کار بدن یک نام بدون معنی مانند foo و پرداختن به ادامه کار است. اما اگر عادت کنید که زمان بیشتری برای فکر کردن در مورد نام خوب اختصاص دهید، به مرور ماهیچه نام‌گذاری را در خود به گونه‌ای خواهید ساخت که به سرعت آن را پرورش دهید. (منظور این است که تمرین در مورد نام‌گذاری همچون تمرین ورزشی که باعث ظاهر شدن عضلات و قوی‌تر شدن آن‌ها می‌شود، ذهن شما را قوی‌تر می‌کند و کافی است تمرین کنید و وقت بیشتری برای انتخاب نام بهتر قرار دهید).

نامهای واقعی را نسبت به نامهای انتزاعی ترجیح دهید



هنگام نام‌گذاری یک متغیر، تابع یا دیگر عناصر بجای نامهای انتزاعی آنها را به شکل به هم پیوسته و واقعی^۱ توصیف کنید. به عنوان مثال در نظر بگیرید که یک متدهای TCP/IP serverCanStart() نامیدهاید(که تست می‌کند ایا سرور می‌تواند روی یک پورت گوش کند یا نه). نام ServerCanStart() تا حدودی انتزاعی است، یک نام دقیق‌تر می‌تواند CanListenOnPort() باشد. این نام مستقیماً توصیف می‌کند که متدهای کاری انجام خواهد داد. در دو مثال بعدی این موضوع را به شکل دقیق‌تری بررسی خواهیم کرد.

DISALLOW_EVIL_CONSTRUCTORS :

این مثال مریوط به کدپایه در گوگل است. در C++ اگر شما یک کپی از سازنده و یا انتساب عملگر^۲ برای کلاس خود تعریف نکنید، یک مقدار پیش فرض اعمال می‌شود. اگر چه این کار مفید است، ولی این متدها به راحتی می‌توانند منجر به نشت حافظه^۳ و خرابکاری‌های دیگری شوند، چرا که در پشت صحنه، در جایی که احتمالاً متوجه آنها نشده‌اید، اجرا شده‌اند.

^۱ concretely

^۲ assignment operator

^۳ memory leaks

به همین دلیل، گوگل با استفاده از یک ماکرو، قانونی برای عدم اجازه به این سازنده‌های شیطانی(evil)^۱ دارد:

```
class ClassName {
private:
    DISALLOW_EVIL_CONSTRUCTORS(ClassName);
public:
    ...
};
```

ماکرو به صورت زیر تعریف شده است:

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
    ClassName(const ClassName&); \
    void operator=(const ClassName&);
```

با قرار دادن این ماکرو در بخش private کلاس، این دو متده به صورت خصوصی شده و بنابراین حتی به صورت تصادفی نیز نمی‌توانند به صورت عمومی مورد استفاده قرار گیرند.

نام DISALLOW_EVIL_CONSTRUCTORS نام خوبی نیست. استفاده از کلمه evil موضوعی بیش از حد جدی را برای یک اشکال قابل بحث به کاربر منتقل می‌کند. از همه مهم‌تر، خیلی شفاف نیست که چه ماکرویی غیر مجاز است. آیا این عبارت، متده() را قبول می‌کند یا نه و اینکه حتی یک سازنده نیز نیست. این نام سال‌ها مورد استفاده قرار می‌گرفت اما در نهایت با چیزی کمتر تحریک کننده و دقیق‌تر بود جایگزین شد:

```
#define DISALLOW_COPY_AND_ASSIGN(ClassName) ...
```

--run_locally:

یکی از برنامه‌های ما، یک پرچم^۲ خط فرمان^۳ اختیاری به نام run_locally داشت. این پرچم باعث می‌شد تا برنامه ضمن چاپ اطلاعات اضافی در مورد اشکال‌زدایی^۴، کنتر اجرا شود. به همین دلیل

^۱ evil

^۲ flag

^۳ Command Line

^۴ Debugging

معمولاً زمانی استفاده می‌شد که تست روی یک دستگاه محلی مثل لپتاپ انجام می‌گرفت و نه زمانی که برنامه روی یک سرور به صورت از راه دور^۱ اجرا می‌شد و کارایی برای ما مهم بود.

شما می‌توانید متوجه شوید که این نام run_locally- از کجا آمده، اما چند اشکال در آن وجود دارد:

- عضو جدید تیم نمی‌داند که این پرچم چه کاری انجام می‌دهد. او هنگام اجرای برنامه به صورت لوکال، از آن استفاده می‌کند اما نمی‌داند چرا چنین چیزی نیاز است.
 - گاهی اوقات نیاز داریم که اطلاعات اشکال‌زدایی را در حین اجرای برنامه به صورت ریموت چاپ کنیم. ارسال run_locally- به یک برنامه که به صورت ریموت در حال اجرا می‌باشد، خنده دار به نظر می‌رسد و فقط گیج کننده است.
 - گاهی اوقات ما در حین گرفتن تست کارایی به صورت لوکال، نمی‌خواهیم که ورود به سیستم کند شود، بنابراین از run_locally- استفاده نمی‌کنیم.
- مشکل این است که run_locally- به دلیل شرایطی که معمولاً از آن استفاده می‌شود، نام گذاری شده است. در عوض، یک پرچم با نامی شبیه extra_logging- مستقیم و صریح‌تر خواهد بود.
- اما اگر run_locally- به انجام کارهای بیشتری از ورود به سیستم نیاز داشته باشد چه باید کرد؟ برای نمونه، فرض کنید که به راه اندازی و استفاده از یک پایگاه داده محلی خاص نیاز داشته باشد. اکنون نام run_locally- و سوشه انگیزتر است چرا که این نام می‌تواند هر دو کار «راه اندازی» و «استفاده از یک پایگاه داده محلی خاص» را به شکل همزمان در خود داشته باشد.
- اما استفاده از این نام برای این اهداف، می‌تواند مبهم و غیرمستقیم باشد. راه حل بهتر این است که پرچم دومی به نام use_local_database-- نیز ایجاد کنیم. حتی اگر مجبور هستید که از دو پرچم استفاده کنید، این کار را انجام دهید، چرا که این پرچم‌ها دقیق‌تر هستند. آن‌ها سعی نمی‌کنند که دو ایده متعامد را در یک کلمه بگنجانند و این اختبار را به شما می‌دهند که در موقع لزوم تنها از یکی از آن‌ها استفاده کنید.

^۱ remote

ضمیمه کردن اطلاعات اضافه به یک نام



همان‌گونه که قبلاً اشاره کردیم، نام یک متغیر شبیه یک کامنت کوچک است. حتی اگر فضای زیادی وجود نداشته باشد، هر اطلاعات اضافی که در یک نام می‌گنجانید، هر بار که یک متغیر دیده می‌شود، این اطلاعات نیز دیده خواهد شد.

بنابراین اگر چیزهایی درباره یک متغیر خیلی مهم بوده و خواننده باید آن‌ها را بداند، ارزش دارد که کلمات بیشتری به نام متغیر اضافه کنید. برای مثال فرض کنید متغیری دارید که شامل یک رشته hexadecimal به صورت زیر است:

```
string id; // Example: "af84ef845cd8"
```

اگر این مهم است که خواننده فرمت ID را به یاد داشته باشد، پس بهتر است از نام `id_hex` استفاده کنید.

مقدارها و واحدها

اگر متغیرهای شما برای اندازه‌گیری هستند (مانند یک لحظه از زمان یا یک تعداد از بایت‌ها)، اینکه واحد آن متغیر را نیز در نام متغیر بگنجانید کمک کننده است.

برای مثال در اینجا یک کد javascript داریم که میزان زمان load یک صفحه وب را اندازه‌گیری می‌کند:

```
var start = (new Date()).getTime(); // top of the page
...
var elapsed = (new Date()).getTime() - start; // bottom of the page
document.writeln("Load time was: " + elapsed + " seconds");
```

هیچ خطای واضحی در این کد وجود ندارد، اما این کد درست کار نمی‌کند. زیرا() زمان را بر اساس واحد میلی ثانیه^۱ بر می‌گرداند و نه ثانیه^۲.

با افزودن _ms به نام متغیرها می‌توانیم همه چیز را دقیق‌تر بیان کنیم:

```
var start_ms = (new Date()).getTime(); // top of the page
...
var elapsed_ms = (new Date()).getTime() - start_ms; // bottom of the page
document.writeln("Load time was: " + elapsed_ms / 1000 + " seconds");
```

علاوه بر واحد زمان، چندین واحد دیگر نیز وجود دارند که در برنامه‌نویسی کاربرد زیادی دارند. در ادامه جدولی از واحدهای استفاده شده در پارامترهای تابع و نسخه‌های بهتری از آن‌ها را می‌توانید ببینید:

پارامترهای تابع	تغییر نام پارامتر به واحد کد شده ^۳
Start(int delay)	delay → delay_secs
CreateCache(int size)	size → size_mb
ThrottleDownload(float limit)	limit → max_kbps
Rotate(float angle)	angle → degrees_cw

^۱ milliseconds

^۲ seconds

^۳ Encoding

کد کردن سایر خصوصیات^۱ مهم

تکنیک ضمیمه کردن اطلاعات بیشتر به یک نام، فقط به مقادیر واحدها محدود نمی‌شود. شما باید هر زمان که مورد چالش برانگیز یا تعجب برانگیزی در مورد متغیر دیدید، از این تکنیک استفاده کنید.

به عنوان مثال، خیلی از سوء استفاده‌های امنیتی ناشی از عدم درک برخی از داده‌هایی است که برنامه شما دریافت کرده و هنوز در وضعیت امن قرار ندارند. برای این کار ممکن است بخواهید از نام متغیرهایی شبیه unsafeMessageBody یا untrustedUrl استفاده کنید. بعد از فراخوانی توابعی که ورودی نامن را حذف می‌کنند، نام متغیر برای نتیجه کار ممکن است trustedUrl یا safeMessageBody باشد.

جدول زیر مثال‌های بیشتری در مورد زمان‌هایی که باید اطلاعات بیشتری در نام گنجانده شود را نشان می‌دهد:

نام مناسب‌تر	نام متغیر	موقعیت
Plaintext_password	Password	یک پسورد به صورت متنی است و باید قبل از پردازش بعدی رمزنگاری شود.
Unscaped_comment	Comment	یک کاربر کامنتی گذاشته است که باید قبل از نمایش escape شود.
Html_utf-8	Html	کدهای html که به UTF-8 تبدیل شده‌اند.
Data_urlenc	Data	داده‌های ورودی در url رمز شده هستند.

شما نباید از خصوصیاتی مانند unescaped_utf-8 یا _utf-8 برای هر متغیری در برنامه خود استفاده کنید. آن‌ها در مکان‌های پر اهمیت قرار دارند که در صورت اشتباه بودن متغیر، یک باگ می‌تواند به آسانی در آن مخفی شده باشد، به خصوص اگر آن باگ امنیتی باشد که پیامدهای آن وخیم است. اساساً اگر یک مورد بحرانی وجود دارد که خواننده باید از آن اطلاع پیدا کند، آن را در نام متغیر لحاظ کنید.

^۱ Attributes

آیا این نماد مجارستانی(Hungarian Notation)^۱ است؟

Hungarian Notation یک سیستم نام گذاری است که به شکل گسترده در شرکت مایکروسافت استفاده شده است. در این سیستم نوع هر متغیر به صورت پیشوند در نام آن اضافه می‌شود. در جدول زیر می‌توانید چند مثال از آن را مشاهده کنید:

نام	معنی
pLast	اشاره‌گر (p) که به آخرین المان یک ساختارداده اشاره می‌کند
pszBuffer	اشاره‌گر (p) برای اشاره به یک (s) string در بافر zero-terminated (z)
Cch	یک شمارنده (c) تعداد کاراکترها (characters (ch)) count
mpcpx	یک (m) از یک اشاره‌گر به یک (pco) color که به یک (px) length با طول متغیر x اشاره می‌کند.

در واقع این استاندارد که نمونه‌ای از ضمیمه کردن خصوصیات به نام‌ها است، سیستمی رسمی و سخت گیرانه است که روی نام گذاری مجموعه، با توجه به خصوصیات خاص آن متمرکز شده است. اما آنچه در این بخش مد نظر ما می‌باشد سیستمی گسترده‌تر و غیر رسمی است: یعنی مشخصه‌های مهم یک متغیر را شناسایی کرده و در صورت لزوم آن‌ها را به طور خوانا نام گذاری کنید. مثلاً به عنوان یک پیشنهاد می‌توانید آن‌ها را یادداشت‌های انگلیسی^۲ بنامید.



^۱ [https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-6.0/aa260976\(v=vs.60\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-6.0/aa260976(v=vs.60)?redirectedfrom=MSDN)

^۲ English Notation

طول یک نام چقدر باید باشد؟

هنگام انتخاب یک نام خوب، این محدودیت ضمنی وجود دارد که نباید آن نام خیلی طولانی باشد. هیچ کسی دوست ندارد با شناسه هایی مانند این کار کند:

```
newINavigationControllerWrappingViewControllerForDataSourceOfClass
```

هرچه یک نام طولانی تر باشد، به خاطر سپردن آن، سخت تر شده و به فضای بیشتری روی صفحه نمایش نیاز خواهد داشت، و گاهی باعث می شود که یک خط به چندین خط اضافی شکسته^۱ شود.

از طرف دیگر برنامه نویس ها می توانند با انتخاب نام های تک کلمه ای، این توصیه را در نظر نگیرند. بنابراین چگونه باید این trade-off را مدیریت کنید؟ چگونه بین نام گذاری یک متغیر به `d`، `days` و `days_since_last_update` تصمیم گیری می کنید؟

بهترین پاسخ دقیقاً به نحوه استفاده از این متغیر بستگی دارد. به همین دلیل و برای کمک به شما چند راهنمایی جهت این تصمیم گیری ارائه شده است.

نام های کوتاه برای قلمروهای^۲ کوتاه مناسب هستند

وقتی به یک تعطیلات کوتاه می روید معمولاً چمدان های کمتری نسبت به تعطیلات طولانی آماده می کنید. به همین ترتیب شناسه هایی که قلمرو کوچکی دارند نیازی به حمل اطلاعات زیاد در نام خود ندارند. یعنی شما می توانید از نام های کوتاه تر برای آن ها استفاده کنید زیرا همه اطلاعات (همچون نوع متغیر، مقداردهی اولیه آن و چگونگی انقضای آن) به راحتی دیده می شود:

```
if (debug) {
    map<string,int> m;
    LookUpNamesNumbers(&m);
    Print(m);
}
```

حتی اگر `m` هیچ اطلاعاتی را در خود جای نداده باشد، مشکلی ایجاد نمی شود، چرا که خواننده در این حال تمام اطلاعات مورد نیاز برای درک این کد را در اختیار دارد.

^۱ wrap

^۲ scope

حال فرض کنید m یک عضو کلاس یا یک متغیر سراسری^۱ بود و شما این قطعه کد را مشاهده می‌کردید:

```
LookUpNamesNumbers(&m);
Print(m);
```

بی‌شک این کد خوانایی کمتری دارد، زیرا مشخص نیست که نوع یا هدف m چیست. بنابراین اگر یک شناسه قلمرو بزرگی داشته باشد، نام آن باید اطلاعات کافی را برای شفاف شدن آن ارائه دهد.

نوشتن نام‌های طولانی-به هر حال مشکلی ندارد

دلایل خوب زیادی برای اجتناب از نام‌های طولانی وجود دارد اما این سخن که «تایپ کردن آن‌ها سخت است» جزو دلایل منطقی محسوب نمی‌شود. هر ویرایشگر متن برنامه‌نویسی قابلیت تکمیل کننده کلمه^۲ را دارد. در کمال تعجب، بعضی از برنامه‌نویسان از این ویژگی اطلاعی ندارند. اگر شما تا کنون این ویژگی را امتحان نکرده‌اید، لطفاً خواندن کتاب را کنار گذاشته و آن را در ویرایشگر کدنویسی خود امتحان کنید:

۱. چند حرف اول نام مد نظر خود را تایپ کنید.
 ۲. دستور تکمیل کننده کلمه را اجرا کنید.
 ۳. اگر نام کامل شده صحیح نیست، دستور را تکرار کنید تا نام صحیح نمایش داده شود.
- بسیار جالب است که این قابلیت روی هر نوع فایل، در هر زبانی برای هر علامتی^۳ کار می‌کند، حتی اگر در حال نوشتن کامنت باشید.

ویرایشگر	دستور
Vi	Ctrl-p
Emacs	Meta-/ (hit ESC, then /)
Eclipse	Alt-/
IntelliJ IDEA	Alt-/
ExtMate	ESC

^۱ global variable

^۲ word completion

^۳ token

۲ مخفف‌ها^۱ و اختصارها

گاه برای کوچک نگه داشتن نام‌ها برنامه‌نویسان به مخفف نویسی و مختصر نویسی متousel می‌شوند، همچون نام‌گذاری یک کلاس با نام BEManager به جای BackEndManager. ولی سوال این جاست که آیا این کوچک شدن‌ها ارزش ایجاد سردرگمی را دارد؟

تجربه به ما ثابت کرده است که معمولاً، اختصار نویسی پروژه‌های خاص، ایده بدی است. این اختصار نویسی‌ها در موقع زیادی برای افراد جدیدی که به پروژه اضافه می‌شوند رمزآلود و ترسناک بوده و حتی با گذشت زمان برای خود نویسنده‌گان نیز رمزآلود و ترسناک به نظر خواهد رسید.

بنابراین قانون اصلی ما این است: آیا یک هم تیمی جدید درک می‌کند که معنای این اسم، چیست؟ اگر پاسخ مثبت است، پس احتمالاً نام مناسبی انتخاب شده است.

برای مثال، برای برنامه‌نویسان عادی است که از eval به جای document، evaluation به جای str به جای string استفاده کنند. بنابراین یک هم تیمی جدید وقتی FormatStr() را می‌بیند به احتمال زیاد متوجه معنی آن می‌شود. با این حال او احتمالاً نمی‌فهمد که معنی BEManager چیست.

دور انداختن کلمات زائد^۲

گاه کلمات داخل یک نام را می‌توان بدون از دست دادن هیچ اطلاعاتی از آن، حذف کرد. برای نمونه به جای ConvertToString() می‌توان از ToString() استفاده کرد، که در عین کوتاهتر بودن هیچ اطلاعاتی را نسبت به کلمه اول از دست نمی‌دهد. به همین ترتیب، استفاده از ServeLoop() به جای DoServeLoop() به همان اندازه شفاف است.

از قالب بندی نام برای انتقال معنی آن استفاده کنید

استفاده از underscore و dash capitalization که به ترتیب خط زیر، خط فاصله و بزرگ نویسی حرف اول کلمه هستند، می‌تواند اطلاعات زیادی را در یک نام قرار دهید. برای مثال در اینجا یک کد C++ را مشاهده می‌کنید که در آن از قراردادهای قالب بندی استفاده شده در پروژه‌های متن باز گوگل استفاده کرده است:

^۱ Acronyms

^۲ Abbreviations

^۳ Unneeded

```

static const int kMaxOpenFiles = 100;
class LogReader {
public:
    void OpenFile(string local_file);
private:
    int offset_;
    DISALLOW_COPY_AND_ASSIGN(LogReader);
};

```

داشتن قالب‌های مختلف برای موجودیت‌های مختلف، مانند نوعی از برجسته کردن سینتکس^۱ است که به شما در خواندن راحت‌تر کد کمک می‌کند.

اکثر قالب‌بندی‌های این مثال بسیار متداول هستند. از قالب‌بندی CamelCase (بزرگ بودن حروف اول کلمات) برای نام کلاس‌ها و از lower_separated برای نام متغیرها استفاده می‌شود.

اما برخی از قراردادها ممکن است شما را متعجب کند، به عنوان نمونه، مقدارهای ثابت^۲ به جای CONSTANT_NAME دارای شکل kConstantName هستند. این استایل از این مزیت برخوردار است که به راحتی از مکروهای #define، که طبق قرارداد به صورت MACRO_NAME هستند، متمازیز می‌شود. متغیرهای عضو کلاس شبیه متغیرهای عادی‌اند با این تفاوت که باید در انتهای آن‌ها یک underscore قرار گیرد، مانند offset_. در ابتدا ممکن است این قرارداد عجیب به نظر برسد اما توانایی تشخیص فوری اعضا از سایر متغیرها را بسیار زیاد می‌کند. به عنوان مثال اگر در حال مشاهده کد یک متده‌لانی به صورت گذرا هستید و این خط را ببینید:

```
stats.clear();
```

ممکن است متعجب شوید که آیا این stats متعلق به این کلاس است؟ آیا این کد، stats داخلی کلاس را تغییر می‌دهد؟ اگر از قرارداد member استفاده شده بود، شما می‌توانستید سریعاً نتیجه بگیرید که stats یک متغیر محلی است. در غیر این صورت باید به صورت stats_ نام‌گذاری می‌شد.

^۱ syntax highlighting

^۲ constant

دیگر قراردادهای قالب بندی^۱

بسته به موضوع پروژه یا زبان برنامه‌نویسی، ممکن است قراردادهای دیگری برای قالب بندی وجود داشته باشد که می‌توانید از آن‌ها برای ایجاد نام‌هایی با اطلاعات بیشتر، استفاده کنید. برای نمونه در JavaScript

کتاب (2008) The Good Parts^۲ (Douglas Crockford, O'Reilly، سازنده‌ها^۳) توابعی که از قبل تعریف شده و هنگام new یا همان ایجاد یک شی فراخوانی می‌شوند) باید با حروف بزرگ (اول کلمات) باشند و توابع معمولی باید با یک حرف کوچک شروع شوند:

```
var x = new DatePicker(); // DatePicker() is a "constructor" function
var y = pageHeight(); // pageHeight() is an ordinary function
```

در اینجا **DatePicker** یک سازنده و **pageHeight** یک تابع عادی است.

اجازه دهید مثال دیگری از JavaScript را با هم ببینیم: زمانی که تابع کتابخانه JQuery را صدای زنید) که نام آن کاراکتر تکی \$ است) یک قرارداد مغاید این است که نتایج JQuery را نیز با یک علامت \$ به صورت پیشوند بکار ببریم:

```
var $all_images = $("img"); // $all_images is a jQuery object
var height = 250; // height is not
```

در این کد واضح است که \$all_images یک شی از نتایج JQuery می‌باشد. به عنوان آخرین مثال کد HTML/CSS زیر را در نظر بگیرید. وقتی به یک تگ HTML یک id یا class اختصاص می‌دهید، هر دو underscore و dash برای مقدار آن‌ها معتبر هستند. یک قرارداد احتمالی این است که از underscore برای جداسازی کلمات در idها و از dash برای جداسازی کلمات در classها استفاده شود:

```
<div id="middle_column" class="main-content"> ...
```

تصمیم به استفاده از چنین قراردادهایی در پروژه‌های خود یا تیم خود به شما بستگی دارد. اما از هر قرارداد دیگری که استفاده می‌کنید، در طول کل پروژه خود به آن پایبند باشید.

^۱ Formatting Conventions

^۲ <http://shop.oreilly.com/product/9780596517748.do>

^۳ constructors

خلاصه

موضوع این فصل به صورت خلاصه، قرار دادن اطلاعات در نام‌های شما بود. هدف از انجام این کار این است که خواننده بتواند اطلاعات زیادی را فقط با خواندن یک نام به دست آورد. نکات خاص پوشش داده شده در این فصل عبارتند از:

- از نام‌های خاص استفاده کنید. برای مثال بسته به شرایطی که دارید ممکن است استفاده از کلماتی شبیه Fetch یا Download به جای Get گزینه‌های بهتری باشد.
- از نام‌های عمومی اجتناب کنید(مانند tmp و retval)، مگر اینکه دلیل خاصی برای استفاده از آنها وجود داشته باشد.
- از نام‌های پیوسته واقعی که چیزی را با جزئیات بیشتری شرح می‌دهند استفاده کنید. نام CanListenOnPort() نسبت به ServerCanStart() واضح‌تر است.
- جزئیات مهم را به نام متغیرها ضمیمه کنید. برای مثال، _ms را به آخر نام متغیری که مقدار آن میلی ثانیه است اضافه کرده و یا عبارت raw را به ابتدای متغیر پردازش نشده که نیاز به عملیات escape دارد، اضافه کنید.
- از نام‌های طولانی برای قلمروهای بزرگ‌تر استفاده کنید. از نام‌های مرموز یک یا دو حرفی برای متغیرهایی که در چندین صفحه از کدها به کار می‌رود، استفاده نکنید. نام‌های کوتاه در موقعي که متغیرها فقط در چند خط از کد (به صورت کوتاه) استفاده می‌شوند، بهتر است.
- از قوانین **underscore**، **capitalization** و مواردی از این دست استفاده کنید. برای مثال می‌توانید علامت _ را به اعضای کلاس اضافه کنید تا آنها را از متغیرهای محلی تمایز دهید.

فصل سوم

نام هایی که نمی توانند متناقض باشند



در فصل قبل، در این مورد که چگونه اطلاعات زیادی را در نام‌ها بگنجانید صحبت کردیم. در این فصل تمرکز ما بر روی موضوعی متفاوت است با این عنوان: مراقب نام‌هایی باشید که می‌توانند اشتباه فهمیده شوند.

کلید طلایی

با پرسیدن سوالاتی از خودتان، نام‌های خود را به دقت بررسی کنید، که دیگر افراد چه معانی دیگری می‌توانند از این نام ببرداشت کنند؟

در این مورد واقعاً سعی کنید خلاق بوده و به طور فعال به دنبال تفاسیر اشتباه^۱ بگردید. این مرحله به شما کمک می‌کند تا نام‌های مبهم را کشف و در نتیجه بتوانید آن‌ها را تغییر دهید.

در مورد مثال‌های این فصل، می‌خواهیم با صدای بلند فکر کنیم، همان‌گونه که در حال مشاهده تعابرات غلط درباره یک نام هستیم، بحث نموده و نام بهتری را انتخاب کنیم.

مثال: Filter()

فرض کنید در حال نوشتن کدی هستید که یک مجموعه از نتایج پایگاه داده را به شکل زیر دستکاری می‌کند:

```
results = Database.all_objects.filter("year <= 2011")
```

به نظر شما نتایج این کد شامل چه مواردی است؟

- اشیائی که مربوط به قبل از سال 2011 هستند؟
- اشیائی که سال آن‌ها قبل از 2011 نیست؟

مشکل این است که filter یک کلمه مبهم است. واضح نیست که آیا معنی آن «انتخاب^۲ کردن» یا «مستثنی^۳ کردن» است. بهتر است از نام filter اجتناب کنید، زیرا به راحتی فهمی اشتباه از آن صورت می‌گیرد.

اگر می‌خواهید «انتخاب» انجام دهید کلمه select نام بهتری است و اگر می‌خواهید «مستثنی کنید» نام بهتر exclude() خواهد بود.

^۱ wrong interpretations

^۲ to pick out

^۳ to get rid of

مثال: Clip(text, length)

فرض کنید تابعی را که محتوای یک پاراگراف را Clip می‌کند به صورت زیر دارید:

```
# Cuts off the end of the text, and appends "..."  
def Clip(text, length):  
    ...
```

به دو روش می‌توانید تصویر کنید که Clip() چگونه رفتار می‌کند:

- این تابع Length را از انتهای متن حذف می‌کند.

- این تابع متن را با حداقل اندازه length کوتاه می‌کند.

هرچند روش دوم(کوتاه سازی)^۱ متحمل‌تر است اما شما هیچ وقت مطمئن نخواهید بود، پس به جای آن که خواننده خود را با تردیدهای ناخوشایند رها کنید، بهتر است تابع را به شکل Truncate(text, length) نام‌گذاری نمایید.

اگرچه هنوز هم نام پارامتر Length قابل سرزنش است و اگر max_length جایگزین آن می‌شد، موضوع را واضح‌تر می‌نمود.

اما هنوز کار ما تمام نشده است. max_length تعبیرهای متعددی را ایجاد می‌کند:

- تعدادی از بایت‌ها^۲

- تعدادی از کاراکترها^۳

- تعدادی از کلمات^۴

همان گونه که در فصل قبل مشاهده نمودید، این از مواردی است که باید واحد آن، به نام اضافه شود و از آنجا که در این مورد منظور تعداد کاراکترها می‌باشد بنابراین به جای max_length باید از max_chars استفاده نماییم.

^۱ Truncation

^۲ Bytes

^۳ Characters

^۴ Words

استفاده از `min` و `max` برای(شامل بودن^۱) محدودیت‌ها

اجازه دهید این گونه بگوییم که برنامه سبد خرید باید از خرید بیش از ده آیتم در یک لحظه توسط افراد جلوگیری کند:

```
CART_TOO_BIG_LIMIT = 10
if shopping_cart.num_items() >= CART_TOO_BIG_LIMIT:
    Error("Too many items in cart.")
```

این کد یک اشکال کلاسیک off-by-one دارد و ما به راحتی می‌توانیم آن را با تغییر `=` به `>` بطرف نماییم:

```
if shopping_cart.num_items() > CART_TOO_BIG_LIMIT:
```

و یا با تعریف مجدد `CART_TOO_BIG_LIMIT` به عدد 11 این اشکال را از بین ببریم. اما مشکل اصلی مبهم بودن نام `CART_TOO_BIG_LIMIT` می‌باشد، چرا که واضح نیست منظور شما «تا آن^۲» یعنی کمتر از آن است یا «تا آن و شامل^۳» یعنی همان کمتر و مساوی آن.

توصیه

شفاف‌ترین راه برای نام گذاری یک محدودیت این است که `min` یا `mx_` را قبل از نام آنچه که قرار است محدود شود، قرار دهید.

در مثال بالا باید `MAX_ITEMS_IN_CART` به عنوان نام انتخاب شود. کد جدید ساده و شفاف است:

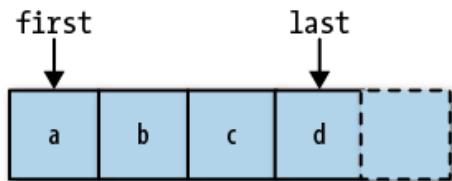
```
MAX_ITEMS_IN_CART = 10
if shopping_cart.num_items() > MAX_ITEMS_IN_CART:
    Error("Too many items in cart.")
```

^۱ Inclusive

^۲ Up to

^۳ Up to and including

ارجحیت first و last برای محدوده‌های^۱ جامع^۲



در اینجا مثال دیگری داریم که شما نمی‌توانید بگویید منظورش از «۲ تا ۴» یعنی فقط ۳ یا ۲ و ۳ یا اینکه شامل ۲ و ۴ نیز می‌شود:

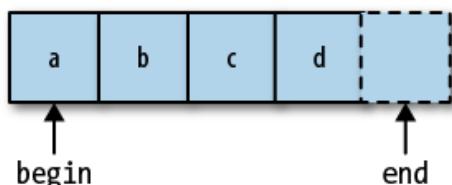
```
print integer_range(start=2, stop=4)
# Does this print [2,3] or [2,3,4] (or something else)?
```

اگرچه نام پارامتر start منطقی به نظر می‌رسد، اما stop می‌تواند به چندین صورت معنی شود. برای محدوده‌های جامع همچون این مثال (یعنی مواردی که در آن محدوده باید شامل دو نقطه آغاز و پایان باشد) گزینه مناسب first/last است:

```
set.PrintKeys(first="Bart", last="Maggie")
```

برخلاف کلمه last، کلمه stop به طور واضح معنی جامعی^۳ دارد. علاوه بر first/last، نامهای min/max نیز احتمالاً برای دامنه‌های جامع^۴ مفید باشند البته با این فرض که در این زمینه درست به نظر می‌رسند.

ارجحیت end و begin برای محدوده‌های جامع/اختصاصی



معمولًا استفاده از محدوده‌های جامع/اختصاصی در عمل راحت‌تر است. به عنوان مثال، اگر شما بخواهید تمام وقایع اتفاق افتاده در ۱۶ اکثیر را چاپ کنید، این ساده‌تر است:

^۱ Ranges

^۲ Inclusive

^۳ inclusive

^۴ inclusive ranges

```
PrintEventsInRange("OCT 16 12:00am", "OCT 17 12:00am")
```

از اینکه بخواهیم بنویسیم:

```
PrintEventsInRange("OCT 16 12:00am", "OCT 16 11:59:59.9999pm")
```

حال این سوال مطرح می‌شود که یک جفت نام خوب برای این پارامترها چیست؟

ظاهرا قرارداد برنامه‌نویسی معمول برای نام گذاری یک محدوده جامع/اختصاصی، کلمات begin/end می‌باشد.

اما اگر بخواهیم دقیق‌تر نگاه کنیم، کلمه end کمی مبهم بوده و به عنوان مثال در جمله « من آخر کتاب هستم^۱ » کلمه end (در زبان انگلیسی) جامع است. متاسفانه زبان انگلیسی کلمه‌ای مختصراً برای عبارت «آخرین مقدار قبلی^۲ » ندارد.

چرا که begin/end خیلی مصطلح هستند و حداقل، از این روش در کتابخانه استاندارد C++ و نیز غالب مواردی که به تکه^۳ شدن یک ارایه نیاز است استفاده می‌شوند. پس ظاهرا این کلمات بهترین گزینه موجود هستند.

نام‌های Boolean

وقتی نامی را برای یک متغیر Boolean انتخاب می‌کنید و یا زمانی که تابعی یک Boolean را برابر می‌گرداند، مطمئن شوید که true و false به طور واضح چه معنایی می‌دهند.

یک مثال چالش برانگیز را ببینید:

```
bool read_password = true;
```

بسته به این که شما آن را چگونه می‌خوانید، می‌توان دو معنای کاملاً متفاوت برداشت کرد:

- ما نیاز داریم که پسورد را بخوانیم
- پسورد همواره خوانده شده است

^۱ I'm at the end of the book

^۲ just past the last value

^۳ sliced

در این مورد، بهتر است از کلمه `read` اجتناب نموده و به جای آن از `need_password` یا `user_is_authenticated` استفاده کنید.

به طور کلی، اضافه کردن کلماتی مانند، `is`، `can`، `has`، `should` معنای نامهای متغیرهای Boolean را شفافتر می‌کند. به عنوان مثال به نظر می‌رسد تابعی با نام `SpaceLeft()` یک مقدار عددی را بر گرداند، ولی اگر قرار باشد که یک مقدار Boolean را برگرداند، بهتر است به شکل `HasSpaceLeft()` نام گذاری شود.

و نکته پایانی این که از عبارت‌های خنثی^۱ در انتخاب یک نام اجتناب کنید، به عنوان مثال به جای:

```
bool disable_ssl = false;
```

بهتر است برای خوانایی ساده‌تر و مختصرتر، از این کد استفاده کنیم:

```
bool use_ssl = true;
```

تطابق انتظارات کاربران^۲

بعضی از نام‌ها به این دلیل که کاربر یک ایده از پیش تعبیر شده از معنای نام را در ذهن خود دارد، گمراх کننده می‌شوند، حتی اگر منظور شما چیز دیگری باشد. در این موارد تسلیم شدن و تغییر نام به چیزی که گمراه کننده نیست، بهترین کار است.

مثال: `get*()`

اکثر برنامه‌نویسان به این قرارداد عادت کرده‌اند که متدهای شروع شده با `get`، به شکل «`lightweight accessors`» هستند و به سادگی یک عضو داخلی را بر می‌گردانند. مخالفت با این قرارداد به احتمال زیاد باعث گمراهی کاربران می‌شود. در اینجا مثالی در زبان Java درباره اینکه چه کاری نباید انجام شود داریم:

```
public class StatisticsCollector {
    public void addSample(double x) { ... }
    public double getMean() {
        // Iterate through all samples and return total / num_samples
    }
    ...
}
```

^۱ negate

^۲ Matching Expectations of Users

در این مورد، پیاده سازی `getMean()` برای تکرار روی داده های گذشته و محاسبه میانگین در پرواز است. اگر داده های زیادی وجود داشته باشد، این مرحله احتمالاً پر هزینه خواهد بود! اما یک برنامه نویس نا آشنا ممکن است با این فرض که معنای آن کم هزینه است، نادانسته `getMean()` فراخوانی کند.

به همین دلیل، متدهای به چیزی شبیه `computeMean()` تغییر نام داده شود که بیشتر نشان دهنده یک عملیات پر هزینه است. البته باید پیاده سازی مجددی صورت گیرد، تا در واقع یک عملیات سبک^۱ محسوب شود.

مثال: `list::size()`

در اینجا یک مثال از کتابخانه استاندارد C++ داریم. کد زیر دلیل سخت پیدا شدن اشکالی بود که باعث می شد یکی از سرورهای ما هنگام crawl کردن بسیار کند شود:

```
void ShrinkList(list<Node>& list, int max_size) {
    while (list.size() > max_size) {
        FreeNode(list.back());
        list.pop_back();
    }
}
```

اشکال این است که خواننده نمی دارد ($O(n)$) یک عملیات `list.size()` است و به جای اینکه فقط یک تعداد از قبل محاسبه شده را برگرداند، نودهای لیست پیوندی را به صورت نود به نود می شمارد و در نتیجه سبب می شود `ShrinkList()` یک عملیات با $O(n^2)$ شود.

کد از نظر تکنیکی درست است و همه تست های واحد^۲ ما را پاس می کند، اما زمانی که `ShrinkList()` روی یک لیست با یک میلیون عنصر صدای شده بود، تمام شدنش حدود یک ساعت طول کشید!

شاید شما بگویید این اشکال تقصیر کسی است که این تابع از کد را فراخوان کرده است، چرا که او باید مستندات را با دقت بیشتری بخواند. این درست است اما در این مورد، این واقعیت که

^۱ lightweight

^۲ Unit test

list.size() یک عمل ثابت-زمان^۱ نیست (به این معنی که مدت زمان اجرای آن ثابت نیست)، تعجب آور است. زیرا دیگر کانتینرها در C++ یک متدهای size() به شکل ثابت-زمان دارند.

Size() باید به صورت countElements() یا countSize() نام گذاری شود که در این صورت احتمال رخداد اشتباه مشابه کمتر خواهد بود. نویسندهای کتابخانه استاندارد C++ احتمالاً می‌خواسته‌اند نام متدهای size() برای همه کانتینرها دیگر مانند vector و map نیز تطابق داشته باشد. اما به دلیل انجام این کار، برنامه‌نویسان به سادگی آن را به عنوان یک عمل سریع اشتباه می‌گیرند، چرا که این روش برای سایر کانتینرها نیز انجام شده است. البته جای نگرانی نبوده و خوشبختانه آخرین استاندارد C++ هم اکنون سرعت اجرای size() را به $O(1)$ کاهش داده است.

wizard کیست؟

خیلی وقت پیش، نویسندهای در حال نصب سیستم عامل OpenBSD بود که در مرحله فرمت کردن دیسک، منوی پیچیده‌ای در مورد پارامترهای دیسک ظاهر شد. یکی از گزینه‌ها حالت جادوگر یا «Wizard mode» بود. او به نظرش رسید که این گزینه کاربر-پسندی است و آن را انتخاب کرد. در کمال ناراحتی برنامه کار را برای دریافت دستورات فرمت کردن دیسک به صورت دستی (در حالت خط-فرمان) قرار داد، که روش روشنی برای خارج شدن از آن وجود نداشت. ظاهراً منظور از wizard این بود که شما جادوگر هستید!

مثال: ارزیابی گزینه‌های مختلف از نامها

هنگام تصمیم گیری در مورد انتخاب یک نام خوب، احتمالاً چندین گزینه را در نظر دارید. این طبیعی است که در مورد معیار انتخاب خود در مورد هر کدام از نامها دچار تردید شوید.

مثال زیر پیچیدگی این فرآیند را نشان می‌دهد:

وبسایتهاي پر ترافيك اغلب از آزمایشات(experiments) برای این تست که آیا یک تغییر در سایت موجب بهبود تجارت‌شان می‌شود یا خیر استفاده می‌کنند. در اینجا مثالی از یک فایل پیکربندی که بعضی از experiments را کنترل می‌کند داریم:

```
experiment_id: 100
description: "increase font size to 14pt"
traffic_fraction: 5%
```

^۱ constant-time

هر آزمایش با حدود ۱۵ جفت صفت/مقدار^۱ تعریف شده است. متناسفانه زمانی که آزمایش دیگری که خیلی شبیه این آزمایش است، تعریف می‌شود، شما مجبور خواهید بود که اکثر این خطوط را کپی کنید:

```
experiment_id: 101
description: "increase font size to 13pt"
[other lines identical to experiment_id 100]
```

فرض کنید ما می‌خواهیم این شرایط را با معرفی روشی برای داشتن یک آزمایش به گونه‌ای درست کنیم که آن آزمایش از ویژگی‌های دیگر آزمایش‌ها مجدد استفاده کند. که در واقع این یک الگوی خواهد بود. نتیجه نهایی چیزی شبیه این خواهید داشت:

```
experiment_id: 101
the_other_experiment_id_I_want_to_reuse: 100
[change any properties as needed]
```

حال سوالی که درباره the_other_experiment_id_I_want_to_reuse مطرح می‌شود این است که واقعاً چه نامی باید برای آن انتخاب شود؟ در اینجا چهار نام در نظر گرفته شده است:

1. Template
2. Reuse
3. Copy
4. Inherit

هر کدام از این نام‌ها برای ما دارای معنایی هستند، زیرا این ما هستیم که این ویژگی جدید را به زبان پیکربندی اضافه می‌کنیم. اما باید تصور کنیم که این نام در نظر کسی که صرفاً کد را می‌بیند و در مورد ویژگی آن چیزی نمی‌داند چگونه به نظر می‌رسد. بنابراین باید هر نام را بررسی کنیم و در مورد راههایی که کسی بتواند آن را اشتباه تفسیر کند فکر کنیم.

.۱. باید تصور کنیم که از نام template استفاده می‌کنیم:

```
experiment_id: 101
template: 100
...
```

^۱ attribute/value

کلمه template دو مشکل دارد. اولاً این در مورد معنایش «من یک template هستم» یا «من از یک template دیگری استفاده می‌کنم» شفاف نیست. دوم اینکه یک template غالباً چیزی انتزاعی است و باید قبل از اینکه واقعی باشد، شرح داده شود. ممکن است کسی فکر کند که یک الگوی آزمایش یک آزمایش «واقعی» نیست. به طور کلی template در این شرایط خیلی مهم است.

۲. حال فرض کنید reuse را انتخاب کنیم:

```
experiment_id: 101
reuse: 100
..
```

کلمه reuse مناسب است اما همان گونه که قبلاً گفتیم ممکن است کسی برداشتش این باشد که این آزمایش حداقل ۱۰۰ مرتبه قابل استفاده مجدد است. تغییر این نام به reuse_id می‌تواند کمک کننده باشد. اما ممکن است دوباره خواننده گیج شده و فکر کند که عبارت reuse_id: 100 به این معنی است که شناسه من برای استفاده مجدد ۱۰۰ است.

۳. اکنون copy را در نظر بگیرید:

```
experiment_id: 101
copy: 100
..
```

کلمه copy کلمه خوبی است، اما به تنها یی ممکن است به نظر برسد ۱۰۰ copy: به این معنی است که این آزمایش ۱۰۰ مرتبه کپی شده است یا این ۱۰۰ امین کپی از چیزی است. برای شفافتر شدن اینکه این عبارت به دیگر آزمایش‌ها اشاره می‌کند، ما باید نام آن را به copy_experiment تغییر دهیم. پس احتمالاً copy_experiment نام تا اینجا کار است.

۴. اما حالا inherit را در نظر بگیرید:

```
experiment_id: 101
inherit: 100
..
```

کلمه inherit برای اکثر برنامه‌نویسان آشنا است و این قابل درک است که اصلاحات بعدی پس از ارثبری^۱ انجام می‌شود. با ارثبری کلاس، شما همه متدها و اعضای کلاسی دیگر را گرفته و آن‌ها

^۱ inheritance

را تغییر می‌دهید و یا چیزی را به کلاس خود اضافه می‌کنید. حتی در زندگی واقعی نیز چنین است، مثلاً وقتی که از کسی چیزی را به ارث می‌برید، این قابل درک خواهد بود که ممکن است آن‌ها را بفروشید یا مالک چیزهای دیگری شوید.

اما اجازه دهید که روشن کنیم که از آزمایش `inherit_from` ارث بری کرده‌ایم. برای این منظور می‌توانیم نام آن را با تغییر به `inherit_from_experiment_id` باحتی `inherit_from` بهبود دهیم.

درکل، `copy_experiment` و `inherit_from_experiment_id` نام‌های خوبی هستند، زیرا باوضوح بیشتری شرح می‌دهند که چه چیزی رخ می‌دهد و در عین حال احتمال سوءتفاهم^۱ در مورد آن‌ها حداقل است.

خلاصه فصل

بهترین نام‌ها، آن‌هایی هستند که سبب سوءتفاهم یا برداشت اشتباه از معنای خود نشود. برای کسی که کد شما را می‌خواند، این نام باید همان معنایی را بدهد که شما مد نظر داشتید و نه چیز دیگر. متناسبانه بسیاری از کلمات انگلیسی وقتی در برنامه‌نویسی استفاده می‌شوند می‌بهم یا دوپهلو هستند، همچون `filter`, `length` و `limit`.

قبل از این که در مورد یک نام تصمیم گیری کنید، کمی آن را به چالش کشیده و تصور کنید چگونه این نام ممکن است باعث سوءتفاهم شود. بهترین نام‌ها در برابر تفاسیر نادرست، مقاوم هستند.

زمانی که از نام‌ها برای تعریف حد بالا و پایین یک مقدار استفاده می‌کنید، `min` و `max` پیش‌سوندهای مناسبی برای استفاده هستند. برای محدوده‌های جامع، `first` و `last` نام‌های خوبی هستند. برای محدوده‌های جامع/اختصاصی^۲ `begin` و `end` بهترین انتخاب هستند زیرا آن‌ها بیشتر مصطلح هستند.

زمانی که یک `Boolean` را نام‌گذاری می‌کنید، برای ایجاد شفافیت بیشتر که این‌ها هستند، از کلماتی مانند `is` و `has` استفاده کرده و از عبارت‌های خنثی اجتناب کنید(همچون `(disable_ssl`)

^۱ misunderstood

^۲ inclusive/exclusive

از انتظارات کاربران درباره معنای کلمات، به شکل دقیق آگاه باشید. برای مثال ممکن است کاربران انتظار داشته باشند که `size()` یا `get()` متدهای سبکی^۱ باشند.

^۱ lightweight

فصل چهارم

زیبا سازی^۱



^۱ Aesthetics

در صفحه آرایی یک مجله موارد بسیاری دخیل است. طول پاراگراف‌ها، اندازه عرض ستون‌ها، ترتیب مقالات و اینکه چه چیزی روی جلد قرار بگیرد. طراحی خوب یک مجله باعث می‌شود که از صفحه‌ای به صفحه دیگر بروید و خواندن آن نیز روان باشد.

سورس کد خوب باید به آسانی به چشم بیاید. در این فصل نشان خواهیم داد که استفاده صحیح از فاصله گذاشتن^۱، تراز بندی^۲ و ترتیب^۳ بخش‌های مختلف چگونه می‌تواند خوانایی کد شما را آسان کند.

به طور خاص، در اینجا سه اصل داریم:

- همواره از طرح بندی(layout) با الگوهایی که خواننده می‌تواند از آن‌ها استفاده کند، استفاده کنید.
- کدهای مشابه را از نظر ظاهری شبیه به هم بنویسید.
- خطوط مرتبط کد را در یک بلوک گروه‌بندی کنید.

زیباسازی^۴ در مقایسه با طراحی

ما در این فصل فقط به دنبال ایجاد یک بهبود ساده در کد با استفاده از روش زیباسازی هستیم. ایجاد این نوع تغییرات، ساده بوده و اغلب خوانایی کد را کمی بهبود می‌دهد. البته زمان‌هایی نیز وجود دارد که یک بازسازی بزرگ می‌تواند کمک بیشتری به شما کند. به نظر ما زیباسازی خوب و طراحی خوب، ایده‌های مستقل هستند که در حالت ایده‌آل باید برای هر دو تلاش کنید.

^۱ spacing

^۲ alignment

^۳ ordering

^۴ aesthetic

چرا زیباسازی^۱ مهم است؟



تصور کنید که مجبور باشید از کلاس زیر استفاده کنید:

```
class StatsKeeper {  
public:  
    // A class for keeping track of a series of doubles  
    void Add(double d); // and methods for quick statistics about them  
private:    int count;           /* how many so    far  
*/ public:  
    double Average();  
private:    double minimum;  
list<double>  
    past_items  
    ;double maximum;  
};
```

^۱ Aesthetics

ب شک زمان مورد نیاز برای فهمیدن این کد نسبت به نسخه مرتب و تمیز شده زیر بسیار بیشتر خواهد بود:

```
// A class for keeping track of a series of doubles
// and methods for quick statistics about them.

class StatsKeeper {
    public:
        void Add(double d);
        double Average();

    private:
        list<double> past_items;
        int count; // how many so far
        double minimum;
        double maximum;
};

};
```

بدیهی است که کار با کدی که از نظر زیبایی ظاهری خوشایندتر باشد، ساده‌تر خواهد بود. اگر درباره آن فکر کنید، متوجه خواهید شد که بیشتر وقت برنامه‌نویسی شما صرف نگاه کردن به کد می‌شود! پس هرچه بتوانید به شکل سطحی کد خود را سریعتر بخوانید، استفاده از آن نیز برای دیگران آسانتر خواهد بود.

تنظيم مجدد خطوط شکسته به صورت استوار^۱ و جمع و جور^۲

فرض کنید که شما در حال نوشتن کد Java برای ارزیابی نحوه رفتار برنامه خود، تحت سرعت‌های مختلف اتصال به شبکه هستید. شما یک TcpConnectionSimulator دارید که در سازنده^۳ خود چهار پارامتر دریافت می‌کند:

۱. سرعت اتصال (Kbps)
۲. میانگین تاخیر^۴ (ms)
۳. میزان jitter^۵ تاخیر (ms)

^۱ Consistent

^۲ Compact

^۳ constructor

^۴ latency

^۵ میزان تغییرات تاخیر پکت‌ها در شبکه

۴. میزان از دست دادن بسته‌ها^۱(درصد)

کد شما سه نمونه^۲ مختلف TcpConnectionSimulator نیاز دارد:

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi = new TcpConnectionSimulator(
        500, /* Kbps */
        80, /* millisecs latency */
        200, /* jitter */
        1 /* packet loss % */);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000, /* Kbps */
            10, /* millisecs latency */
            0, /* jitter */
            0 /* packet loss % */);

    public static final TcpConnectionSimulator cell = new TcpConnectionSimulator(
        100, /* Kbps */
        400, /* millisecs latency */
        250, /* jitter */
        5 /* packet loss % */);
}
```

این مثال به تعداد زیادی خطوط کد اضافی محتاج است تا داخل یک محدودیت 80 کاراکتری (که استاندارد کدنویسی در شرکت است) قرار بگیرد. متاسفانه این امر باعث شده که تعریف t3_fiber نسبت به خطوط دیگر متفاوت به نظر رسیده و بدون هیچ دلیلی سبب جلب توجه کلمه t3_fiber می‌شود(اگر با دقیق بودن نگاه کنید متوجه می‌شوید که عبارت new TcpConnectionSimulator در تعریف t3_fiber به خط بعدی رفته است).

^۱ Packet lost

^۲ instance

این کد از اصلی که می‌گوید کدهای مشابه باید شبیه هم به نظر برسند، پیروی نمی‌کند. برای رعایت این اصل، می‌توانیم در هر نمونه بعد از علامت = یک Enter بزنیم تا کد جدید به شکل زیر شود:

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(
            500, /* Kbps */
            80, /* millisecs latency */
            200, /* jitter */
            1 /* packet loss % */);
    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000, /* Kbps */
            10, /* millisecs latency */
            0, /* jitter */
            0 /* packet loss % */);
    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(
            100, /* Kbps */
            400, /* millisecs latency */
            250, /* jitter */
            5 /* packet loss % */);
}
```

این کد از نظر الگو، زیبایی خوبی داشته و خوانایی آن در یک نگاه، ساده‌تر است، اما متناسبانه از فضای عمودی زیادی استفاده می‌کند. همچنین هر کامنت سه مرتبه تکرار شده است.

یک راه جمع و جورتر نشان دادن این کلاس به این شکل است که:

```
public class PerformanceTester {
    // TcpConnectionSimulator(throughput, latency, jitter, packet_loss)
    // [Kbps] [ms] [ms] [percent]
    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(500,     80,      200,      1);
    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(45000,   10,       0,       0);
    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(100,     400,     250,      5);
}
```

ما کامنت‌ها را در قسمت بالا و همه پارامترها را در یک خط قرار داده‌ایم. حال، اگرچه این کامنت درست در جلوی شما نیست اما در عوض برنامه با خطوط جمع و جورتری به شکل ستون‌های یک جدول بصورت مرتب ارائه شده است.

از متدها برای پاک کردن بی نظمی استفاده کنید

فرض کنید یک دیتابیس پرسنلی دارید که تابع زیر را ارائه می‌دهد:

```
// Turn a partial_name Like "Doug Adams" into "Mr. Douglas Adams".
// If not possible, 'error' is filled with an explanation.
string ExpandFullName(DatabaseConnection dc, string partial_name, string* error);
```

و این تابع توسط تعدادی از مثال‌ها تست می‌شود:

```
DatabaseConnection database_connection;
string error;
assert(ExpandFullName(database_connection, "Doug Adams", &error)
      == "Mr. Douglas Adams");
assert(error == "");
assert(ExpandFullName(database_connection, " Jake Brown ", &error)
      == "Mr. Jacob Brown III");
assert(error == "");
assert(ExpandFullName(database_connection, "No Such Guy", &error) == "");
assert(error == "no match found");
assert(ExpandFullName(database_connection, "John", &error) == "");
```

```
assert(error == "more than one result");
```

این کد از نظر زیبایی ظاهری خوشایند نیست. برخی خطوط بسیار طولانی بوده و ادامه آن‌ها به خط بعدی رفته است و همچنین ظاهر کد زشت بوده و الگوی مرتبی در آن وجود ندارد. هرچند این مورد تنها با تنظیم مجدد خطوط شکسته قابل حل است ولی مشکل بزرگتر این است که تعداد زیادی از رشته‌ها شبیه «assert(ExpandFullName(database_connection))» تکرار شده‌اند و عبارت «error» نیز به همین شکل است.

برای بهتر کردن این کد، نیازمند یک متدهستیم تا کد بتواند به شکل زیر نمایش داده شود:

```
CheckFullName("Doug Adams", "Mr. Douglas Adams", "");  
CheckFullName(" Jake Brown ", "Mr. Jake Brown III", "");  
CheckFullName("No Such Guy", "", "no match found");  
CheckFullName("John", "", "more than one result");
```

اکنون این کد در نشان دادن اینکه چهار تست آن هم با پارامترهای متفاوت در حال رخ دادن است، شفافتر به نظر می‌رسد. حتی اگر کارهای کثیف^۱ داخل تابع CheckFullName() باشد، این تابع چندان هم بد نخواهد بود:

```
void CheckFullName(string partial_name,  
  
    string expected_full_name,  
    string expected_error) {  
  
    // database_connection is now a class member  
    string error;  
  
    string full_name = ExpandFullName(database_connection, partial_name, &error);  
    assert(error == expected_error);  
    assert(full_name == expected_full_name);  
}
```

اگر هدف ما فقط این باشد که کد را از نظر زیباسازی خوشایندتر کنیم، این تغییر مزایای جانبی دیگری نیز دارد:

- سبب از بین رفتن کدهای تکراری زیادی نسبت به قبل شده که نتیجه آن جمع و جورتر شدن کد خواهد بود.

^۱ dirty work

- مهمترین بخش از هر مورد تست^۱ یعنی نام‌ها و رشته‌های خط^۲ اکنون به طور واضح و در یک سمت مشخص قرار گرفته است.
 - قبل این رشته‌ها با علامت‌هایی^۳ مانند database_connection و error پراکنده بودند که سبب سخت شدن درک آن‌ها با یک نگاه می‌شد.
 - اکنون باید افزودن تست‌های جدید ساده‌تر باشد.
 - نکته اخلاقی داستان این است که زیبا کردن یک کد، اغلب به چیزی بیشتر از یک بهبود ظاهری، منجر می‌شود و این کار ممکن است به بهتر شدن ساختار کد شما نیز کمک کند.
- در صورت مفید بودن از ترازبندی ستونی^۴ استفاده کنید**

در یک نگاه گوشه‌ها و ستون‌های صاف خوانایی راحت‌تری را برای خواننده به همراه دارد. گاهی اوقات می‌توانید «ترازبندی ستون» را برای ساده‌تر کردن خواندن کد، معرفی کنید. به عنوان مثال در بخش قبلی شما می‌توانستید از فضای خالی بعد از کلمات در آرگومان‌های تابع CheckFullName() استفاده کنید:

```
CheckFullName("Doug Adams"    , "Mr. Douglas Adams" , "");  
CheckFullName(" Jake Brown ", "Mr. Jake Brown III", "");  
CheckFullName("No Such Guy" , ""                  , "no match found");  
CheckFullName("John"        , ""                  , "more than one result");
```

در این کد، تمایز قائل شدن بین آرگومان‌های دوم و سوم تابع CheckFullName() ساده‌تر است.

در اینجا مثالی ساده با یک گروه بندی بزرگ از تعریف متغیرها را بررسی می‌کنیم:

```
# Extract POST parameters to local variables  
details  = request.POST.get('details')  
location = request.POST.get('location')  
phone    = request.POST.get('phone')  
email   = request.POST.get('email')  
url     = request.POST.get('url')
```

^۱ test case

^۲ error strings

^۳ tokens

^۴ Column Alignment

همان گونه که احتمالا متوجه شده‌اید، تعریف سوم یک اشتباه تایپی دارد(به جای کلمه request نوشته شده است). اشتباهات این چنینی وقتی همه چیز به صورت مرتب نوشته شده باشد، به شکل برجسته‌تری ظاهر می‌شوند.

در کد پایه دستور wget، گزینه‌های موجود در خط فرمان(بیشتر از ۱۰۰ مورد از آن‌ها) به شکل زیر لیست شده‌اند:

```
commands[] = {
    ...
    { "timeout",           NULL,           cmd_spec_timeout },
    { "timestamping",     &opt.timestamping, cmd_boolean },
    { "tries",             &opt.ntry,        cmd_number_inf },
    { "useproxy",          &opt.use_proxy,   cmd_boolean },
    { "useragent",         NULL,           cmd_spec_useragent },
    ...
};
```

این شیوه باعث شده است که این لیست به راحتی و در یک نگاه قابل خواندن بوده و پریدن از یک ستون به ستون بعدی خیلی ساده باشد.

آیا باید از ترازبندی ستونی استفاده کرد؟

لبه‌های ستون مانند نزددهای راه پله هستند که دنبال کردن کد را آسانتر می‌کنند.(مانند زمانی که از پله‌ها بالا می‌روید و نزددها را با دست خود می‌گیرید). این یک مثال خوب از «کدهای مشابه را شبیه هم بسازید» است.

اما بعضی از برنامه‌نویسان این شیوه را نمی‌پسندند. یک دلیل این است که این شیوه به کار بیشتری جهت راه اندازی و حفظ ترازبندی نیاز دارد. دلیل دیگر این است که این شیوه هنگام ایجاد تغییرات «تفاوت^۱» بزرگ را ایجاد می‌کند چرا که اگر یک خط تغییر کند احتمالا سبب تغییر پنج خط دیگر می‌شود که این تغییر در اکثر موارد به دلیل وجود فضای خالی است.

پیشنهاد ما این است که این شیوه را امتحان کنید. در تجربه ما، آن قدر هم که برنامه‌نویسان می‌ترسند هم طول نمی‌کشد و اگر احیاناً این کار انرژی و وقت زیادی از شما گرفت به راحتی می‌توانید آن را ادامه ندهید.

^۱ diff

یک ترتیب معنا دار^۱ انتخاب و از آن به طور مداوم استفاده کنید

موارد زیادی وجود دارد که ترتیب کد روی درستی^۲ آن اثر نمی‌گذارد، به عنوان مثال، تعریف پنج متغیر زیر به هر ترتیبی می‌تواند نوشته شود:

```
details = request.POST.get('details')
location = request.POST.get('location')
phone = request.POST.get('phone')
email = request.POST.get('email')
url = request.POST.get('url')
```

در شرایط مشابه، اینکه آنها را به یک ترتیب معنادار و نه صرفاً به شکل تصادفی بنویسید، مفید خواهد بود. چند ایده برای این کار وجود دارد:

- ترتیب متغیرها را با ترتیب فیلدهای <input> مربوط به فرم HTML مطابقت دهید.
- آنها را از «اهمیت بیشتر» به «اهمیت کمتر» مرتب کنید.
- آنها را به صورت «ترتیب حروف الفبا» مرتب کنید.

ولی به یاد داشته باشید که ترتیب به هر صورتی که باشد، باید از همان ترتیب در سراسر کد خود استفاده کنید، زیرا تغییر ترتیب در بخش‌های دیگر کد، می‌تواند باعث سردرگمی خواننده شود:

```
if details: rec.details = details
if phone:    rec.phone     = phone      # Hey, where did 'location' go?
if email:    rec.email    = email
if url:      rec.url      = url
if location: rec.location = location   # Why is 'location' down here now?
```

اعلان‌ها^۳ را در یک بلوک قرار دهید

از آنجا که مغز انسان به طور طبیعی به ساختار و قواعد گروه بندی‌ها و سلسله مراتب آنها دقت می‌کند، می‌توانید با ساماندهی ساختار به خواننده کمک کنید تا کدهای شما را سریع‌تر و به شکل

^۱ Meaningful Order

^۲ Correctness

^۳ Declarations

خلاصه درک کند. به عنوان مثال در اینجا یک کلاس C++ برای فرانت‌اند یک سرور با تمام اعلان‌های متدهای داریم:

```
class FrontendServer {
public:
    FrontendServer();
    void ViewProfile(HttpRequest* request);
    void OpenDatabase(string location, string user);
    void SaveProfile(HttpRequest* request);
    string ExtractQueryParam(HttpRequest* request, string param);
    void ReplyOK(HttpRequest* request, string html);
    void FindFriends(HttpRequest* request);
    void ReplyNotFound(HttpRequest* request, string error);
    void CloseDatabase(string location);
    ~FrontendServer();
};

};
```

هرچند این کد وحشتناک نیست، اما مطمئناً طرح‌بندی آن کمکی به خواننده برای درک همه متدها با یک نگاه نمی‌کند. به جای قرار دادن متدها در یک بلوک غول پیکر، باید از نظر منطقی این گونه گروه‌بندی شوند:

```
class FrontendServer {
public:
    FrontendServer();
    ~FrontendServer();
    // Handlers
    void ViewProfile(HttpRequest* request);
    void SaveProfile(HttpRequest* request);
    void FindFriends(HttpRequest* request);
    // Request/Reply Utilities
    string ExtractQueryParam(HttpRequest* request, string param);
    void ReplyOK(HttpRequest* request, string html);
    void ReplyNotFound(HttpRequest* request, string error);
    // Database Helpers
    void OpenDatabase(string location, string user);
    void CloseDatabase(string location);
```

^۱ method declarations

{};

این نسخه از کد خیلی راحت‌تر و در یک نگاه درک شده و برای خواندن نیز ساده‌تر است، حتی اگر خطوط کد بیشتری وجود داشته باشد، چرا که شما سریعاً متوجه وجود چهار بخش با اهمیت بالاتر شده و سپس هر زمان که لازم شد جزئیات هر بخش را می‌خوانید.

تقسیم‌بندی کد به صورت پاراگراف‌ها^۱

یک متن نوشتاری به چندین دلیل به پاراگراف‌های جداگانه تقسیم می‌شود:

- این کار راهی برای گروه‌بندی ایده‌های مشابه با یکدیگر و جدا کردن شان از دیگر ایده‌ها است.
- این کار یک رد پای^۲ بصری فراهم می‌کند که بدون آن، به راحتی ممکن است مکان خود را در صفحه گم کنید.
- این کار رفتن از یک پاراگراف به پاراگراف دیگر را تسهیل می‌کند.

یک کد برنامه نویسی شده نیز به دلایل مشابه باید به پاراگراف‌هایی تقسیم شود. به عنوان مثال، کسی علاقه‌مند به خواندن یک کد عظیم مانند این نیست:

```
# Import the user's email contacts, and match them to users in our system.

# Then display a list of those users that he/she isn't already friends with.

def suggest_new_friends(user, email_password):

    friends = user.friends()

    friend_emails = set(f.email for f in friends)

    contacts = import_contacts(user.email, email_password)

    contact_emails = set(c.email for c in contacts)

    non_friend_emails = contact_emails - friend_emails

    suggested_friends = User.objects.select(email__in=non_friend_emails)

    display['user'] = user
```

^۱ Paragraphs

^۲ stepping stone

```

display['friends'] = friends

display['suggested_friends'] = suggested_friends

return render("suggested_friends.html", display)

```

هرچند این کد واضح نیست، اما از آنجا که این تابع چندین مرحله متمایز را طی می‌کند، تفکیک خطوط آن توسط پاراگراف‌ها، برای درک آن بسیار مفید خواهد بود:

```

def suggest_new_friends(user, email_password):
    # Get the user's friends' email addresses.
    friends = user.friends()
    friend_emails = set(f.email for f in friends)
    # Import all email addresses from this user's email account.
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)
    # Find matching users that they aren't already friends with.
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)
    # Display these lists on the page.
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends
    return render("suggested_friends.html", display)

```

نکته‌ای که ما آن را در بخش خلاصه فصل نیز بیان خواهیم کرد این است که هر پاراگراف را کامنت گذاری کنید، این کار سبب می‌شود خواننده به راحتی و در یک نگاه کد را درک کند(فصل ۵ «دانستن اینکه چه چیزی را کامنت کنیم» را مشاهده کنید).

همچون یک متن نوشتاری، ممکن است چندین روش برای تقسیم‌بندی کدها وجود داشته باشد و هر برنامه‌نویسی ممکن است پاراگراف‌های کوچکتر یا بزرگتر را ترجیح دهد.

سبک شخصی یا قوانین و قواعد!

گزینه‌های زیباسازی خاصی وجود دارد که فقط در استایل شخصی مهم‌اند. برای نمونه جایی که برآکت برای تعریف یک کلاس باز شده است، به این شکل:

```
class Logger {  
    ...  
};
```

و یا به این شکل باشد:

```
class Logger {  
    ...  
};
```

اگرچه ترجیح دادن یکی از این استایل‌ها به میزان قابل توجهی روی خوانایی کدپایه^۱ اثر نمی‌گذارد، ولی اگر این دو استایل هم زمان و به شکل مخلوط در کل کد استفاده شوند، روی خوانایی کد اثر خواهد گذاشت.

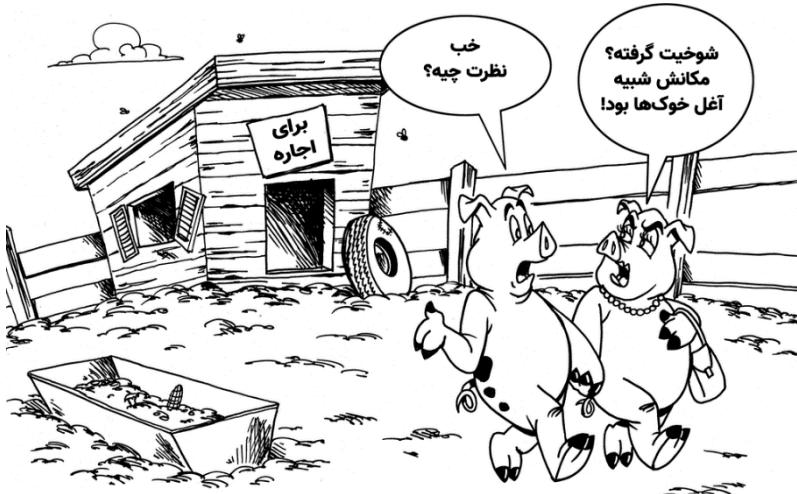
هر چند ما پروژه‌های زیادی را با این احساس که شبیه تیمی هستیم که از استایل اشتباه استفاده می‌کند، انجام داده‌ایم، ولی در عین حال قواعد^۲ پروژه را دنبال می‌کردیم، چرا که می‌دانستیم قواعد تیم بسیار مهم‌تر هستند.

^۱ Codebase

^۲ Consistency

کلید طلازی

استایل‌های همیشگی از استایل «مناسب^۱» بهتر هستند.

**خلاصه فصل**

هر کسی ترجیح می‌دهد کدهایی را که از نظر زیبایی ظاهری خوشایند هستند بخواند. با قالب بندی^۲ کدهای خود به صورت مداوم، به روشنی معنادار، خواندن آن را ساده‌تر و سریعتر می‌کنید.

در اینجا تکنیک‌های خاصی که در طول این فصل به آن‌ها پرداختیم آورده شده است:

- اگر چندین بلوک از کد، کاری مشابه را انجام می‌دهند، سعی کنید آن‌ها را از حیث زیباسازی به یک شکل بنویسید.
- ترازبندی^۳ کد در ستون‌هائے می‌تواند نگاه گذرا به کد را ساده‌تر کند. اگر کد در جایی به A، B و C اشاره می‌کند، در مکان دیگر آن را به صورت B، C و A ننویسید. ترتیبی مشخص و معنی‌دار انتخاب کرده و به آن پایبند باشید.

از خطوط خالی برای تقسیم‌بندی بلوک‌های بزرگ به پاراگراف‌های منطقی استفاده کنید.

^۱ Right

^۲ formatting

^۳ Aligning

^۴ columns

فصل پنجم

چه چیزی را کامنت کنیم

کتابچه راهنمای استفاده از محصول



هدف این فصل کمک به برای فهمیدن این نکته است که «چه چیزی را باید کامنت کنید». احتمالاً می‌گویید هدف از نوشتن کامنت، توضیح این است که کد چه کاری انجام می‌دهد، اما این تنها بخش کوچکی از فواید کامنت نوشتن است.

کلید طلایی

هدف از نوشتن کامنت کمک به خواننده است تا به همان اندازه که نویسنده کد آن را درک کرده، خواننده نیز کد را بفهمد.

هنگامی که در حال نوشتن کد هستید، اطلاعات با ارزش زیادی در مغزتان دارید، ولی زمانی که دیگران کد شما را می‌خوانند، این اطلاعات با ارزش وجود ندارند. تنها چیزی که آن‌ها دارند، کدی است که جلوی آن‌ها قرار دارد.

در این فصل مثال‌های زیادی را در این مورد که چه زمانی این اطلاعات داخل مغزتان را به شکل کامنت بنویسید، به شما نشان خواهیم داد. البته به مواردی که جذابیت کمتری دارند، نمی‌پردازیم، در عوض بر روی جنبه‌های جالب‌تر و «مورد توجه قرار نگرفته^۱» از کامنت گذاری متمرکز می‌شویم.

در این فصل به سه بخش مهم خواهیم پرداخت:

- دانستن اینکه چه چیزی را کامنت نکنیم.
- ثبت افکارتان همانند کدنویسی‌تان.
- قرار دادن خودتان به جای خواننده، تا تصور کنید که چه چیزی را باید بداند.

^۱ Undeserved

چه چیزی را نباید کامنت کنیم



از آنجا که خواندن یک کامنت زمان زیادتری از خواندن کد برد و فضایی را در بالای صفحه اشغال می‌کند، بهتر است چیزی که مد نظر دارید ارزش کامنت نوشتن را داشته باشد. حال این سوال مطرح می‌شود که چگونه می‌توان مرز بین یک کامنت بی ارزش و یک کامنت خوب را فهمید؟

تمام کامنت‌های کد زیر بی ارزش هستند:

```
// The class definition for Account
class Account {
    public:
        // Constructor
        Account();
        // Set the profit member to a new value
        void SetProfit(double profit);
        // Return the profit from this Account
        double GetProfit();
};
```

زیرا هیچ اطلاعات جدیدی را ارائه نداده و هیچ کمکی برای درک بهتر کد به خواننده نمی‌کنند.

کلید طلایی

در مورد حقایقی که خودشان می‌توانند سریعاً با خواندن کد به دست آیند، کامنت ننویسید.

البته استفاده از این کلید طلایی نسبی است، به عنوان مثال کامنت زیر را که برای کد پایتون نوشته شده است در نظر بگیرید:

```
# remove everything after the second '*'
name = '*'.join(line.split('*')[2])
```

از نظر فنی، این کامنت هیچ اطلاعات جدیدی را ارائه نداده و اگر خودتان به کد نگاه کنید، در نهایت متوجه خواهید شد که چه کاری انجام می‌دهد. اما برای اکثر برنامه‌نویسان، خواندن کد کامنت گذاری شده خیلی سریعتر از فهمیدن کد بدون کامنت است.

فقط برای اینکه کدتان کامنت داشته باشد! کامنت گذاری نکنید



بعضی از اساتید، دانشجویان را ملزم به کامنت گذاری برای هر تابع در تکالیف کدنویسی خود می‌کنند. در نتیجه بعضی از برنامه‌نویسانها در مورد رها کردن یک تابع بدون داشتن کامنت احساس گناه کرده و در پایان، اسم توابع و آرگومان‌های آن را در یک جمله به صورت کامنت بازنویسی می‌کنند:

^۱ Quickly

```
// Find the Node in the given subtree, with the given name, using the given depth.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

همان‌گونه که مشاهده می‌کنید کامنت‌های این کد در دسته کامنت‌های بی ارزش قرار می‌گیرد چرا که تعریف تابع و کامنت نوشته شده تقریباً یکسان هستند. برای بهبود این کد، این کامنت باید حذف شود.

اگر می‌خواهید کامنتی در اینجا داشته باشید، می‌توانید در مورد جزئیات مهم‌تر نیز توضیح دهید:

```
// Find a Node with the given 'name' or return NULL.
// If depth <= 0, only 'subtree' is inspected.
// If depth == N, only 'subtree' and N Levels below are inspected.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

برای نام‌های بد کامنت ننویسید، در عوض، نام‌های بهتری انتخاب کنید

لازم نیست یک کامنت، بد بودن یک نام را جبران کند. برای مثال، در اینجا یک کامنت بی فایده برای تابعی به نام `CleanReply()` نوشته شده است:

```
// Enforce limits on the Reply as stated in the Request,
// such as the number of items returned, or total byte size, etc.
void CleanReply(Request request, Reply reply);
```

بخش اعظم این کامنت در حال توضیح این مطلب است که معنی `clean` چیست. کافی است به جای آن عبارت «`enforce limits`» به نام تابع اضافه شود:

```
// Make sure 'reply' meets the count/byte/etc. limits from the 'request'
void EnforceLimitsFromRequest(Request request, Reply reply);
```

نام جدید این تابع اطلاعات مستند زیادی در خودش دارد. به یاد داشته باشید که یک نام خوب، بهتر از یک کامنت خوب است زیرا نام تابع در هرجایی که تابع استفاده شده باشد، دیده می‌شود.

در ادامه مثال دیگری از استفاده کامنت به دلیل انتخاب یک نام ضعیف برای یک تابع را مشاهده می‌کنید:

```
// Releases the handle for this key. This doesn't modify the actual registry.
void DeleteRegistry(RegistryKey* key);
```

به نظر می‌رسد نام DeleteRegistry() شبیه یک تابع خطرناک است (این تابع حافظه registry را delete می‌کند!). کامنت نوشته شده به معنی «در واقع این رجیستری را تغییر نمی‌دهد» است و تلاش می‌کند که از گنج بودن نام تابع، رفع ابهام کند.

به جای آن می‌توانیم از یک نام خود-مستندساز^۱ بهتر به صورت زیر استفاده کنیم:

```
void ReleaseRegistryHandle(RegistryKey* key);
```

به طول کلی، شما نمی‌خواهید کامنت‌های crutch^۲ داشته باشید. کامنت‌هایی هستند که سعی می‌کنند ناخوانا بودن کد شما را جبران کنند. کدنویس‌ها اغلب این وضعیت را به شکل یک قانون، این گونه بیان می‌کنند که: «کد خوب بهتر از کد بد + کامنت خوب^۳» است.

ثبت افکار خود

حال که فهمیدید چه چیزی را کامنت نکنید، بباید در این مورد که چه چیزی را باید کامنت کنیم، بحث کنیم.

اکثر کامنت‌های خوب می‌توانند به سادگی از جمله «ثبت افکار خود» به دست آیند، که شامل مهمترین افکار شما، هنگام نوشتن کد می‌باشند.

افزودن «توضیحات کارگردان»

فیلم‌ها اغلب بخشی با عنوان «توضیحات کارگردان» دارند که در آن فیلم‌ساز بینش خود را ارائه داده و داستان‌هایی را برای کمک به فهمیدن اینکه این فیلم چطور ساخته شده است، بیان می‌کند. به طور مشابه، شما نیز باید کامنت‌ها را برای ثبت دیدگاه‌های با ارزش درباره کد اضافه کنید.

به عنوان مثال این کامنت:

```
// Surprisingly, a binary tree was 40% faster than a hash table for this data.  
// The cost of computing a hash was more than the left/right comparisons.
```

چیزهایی را به خواننده یاد داده و از هرگونه اتلاف وقت آنان جلوگیری می‌کند.

^۱ self-documenting

^۲ یک اصطلاح به معنی عصایی کمکی زیر بغل داشتن است. در اینجا منظور این است که کامنت به عنوان عصایی کمکی برای فهمیدن کد به خواننده استفاده شود.

^۳ good code > bad code + good comments

به عنوان مثال دیگر کامنت زیر را در نظر بگیرید:

```
// This heuristic might miss a few words. That's OK; solving this 100% is hard.
```

بدون نوشتن این کامنت، خواننده ممکن است فکر کند که یک اشکال وجود داشته و بخشی از زمان خود را هنگام تلاش برای تست مواردی که باعث شکست کد می‌شود، تلف کند و یا حتی سعی کند باگ آن را رفع نماید.

همچنین یک کامنت می‌تواند دلیل عالی نبودن شکل ظاهری کد را شرح دهد:

```
// This class is getting messy. Maybe we should create a 'ResourceNode' subclass to
// help organize things.
```

این کامنت می‌گوید که کد ما کثیف است اما همچنان نفر بعدی را برای ترمیم آن تشویق می‌کند (با ارائه مشخصاتی در مورد چگونگی بهبود آن).

با نبود هیچ کامنتی، خیلی از خواننده‌ها با دیدن کدهای کثیف دچار ترس شده و از دست زدن به آن‌ها خودداری می‌کنند.

نقص‌های موجود در کد خود را کامنت کنید

کد به طور مداوم در حال تغییر و تحول است و در طی این مسیر حتماً نقص‌هایی خواهد داشت. از مستند کردن این نقص‌ها خجالت نکشید. مثلاً، زمانی که کد نیازمند به بهبودی است می‌توانید از عبارت زیر استفاده کنید:

```
// TODO: use a faster algorithm
```

یا هنگامی که کد کامل نشده است:

```
// TODO(dustin): handle other image formats besides JPEG
```

برخی از نشانه‌هایی که بین برنامه‌نویسان رایج و محبوب شده‌اند را می‌توانید مشاهده کنید:

نشانه	معنی عمومی
TODO:	برخی کارهایی که باید انجام/اضافه شود
FIXME:	این بخش از کد نیازمند اصلاح است
HACK	استفاده از ترقیت هوشمندانه برای حل مشکل
XXX	هشدار! یک مشکل اساسی وجود دارد

ممکن است تیم شما قراردادی برای زمان یا شرط استفاده از این نشانه‌ها داشته باشد. به عنوان مثال: TODO ممکن است برای نشان دادن توقف اشکالات^۱ رزرو شده باشد. اگر چنین است، پس برای نقص‌های جزئی‌تر می‌توانید بجای TODO از چیزی شبیه todo: (با حروف کوچک) و یا برای استفاده maybe-later:

نکته مهم این است که شما باید همیشه در مورد افکار خود، و نیز در این مورد که کد باید در آینده چه تغییری پیدا کند، راحت کامنت بنویسید. کامنت‌هایی شبیه این عبارات، به خوانندگان بینش بهتری در مورد کیفیت و وضعیت کد داده و حتی ممکن است در مورد چگونگی بهبود کد نیز ایده‌هایی به آن‌ها بدهد.

در مورد ثابت‌ها^۲ کامنت بنویسید

هنگام تعریف یک ثابت، غالباً این سوال وجود دارد که چرا این متغیر ثابت، مقدار خاصی دارد و مقدار خاص آن چیست؟ به عنوان مثال ممکن است چنین ثابتی را در کد خود بینید:

```
NUM_THREADS = 8
```

به نظر نمی‌رسد که این خط، نیازمند کامنت باشد، اما به احتمال زیاد برنامه‌نویسی که آن را انتخاب کرده است اطلاعات بیشتری در مورد آن می‌داند:

```
NUM_THREADS = 8 # as long as it's >= 2 * num_processors, that's good enough.
```

با این کامنت، شخصی که کد را می‌خواند یک راهنمای در مورد نحوه تنظیم این متغیر ثابت دارد (به عنوان مثال، تنظیم آن به مقدار ۱ یا ۵۰ احتمالاً به ترتیب خیلی کم یا بیش از حد زیاد است).

گاهی اوقات مقدار دقیق یک ثابت به هیچ وجه مهم نیست که در این موارد، وجود یک کامنت، مفید به نظر می‌رسد:

```
// Impose a reasonable limit - no human can read that much anyway.
const int MAX_RSS_SUBSCRIPTIONS = 1000;
```

گاهی نیز این ثابت دارای مقداری است که بسیار دقیق تنظیم شده است و احتمالاً نباید خیلی تغییر داده شود:

^۱ Issues

^۲ Constants

```
image_quality = 0.72; // users thought 0.72 gave the best size/quality tradeoff
```

در همه این مثال‌ها ممکن است شما به اضافه کردن کامنت فکر نکرده باشید، اما این کار، کاملاً مفید است. برخی از ثابت‌ها هستند که نیاز به کامنت ندارند، زیرا نام آن‌ها به اندازه کافی واضح است (مثلا: SECONDS_PER_DAY). اما تجربه به ما ثابت کرده است که بیشتر ثابت‌ها را می‌توان با افزودن کامنت بهبود داد. مسئله این است که شما در زمان تصمیم‌گیری در مورد مقداردهی این ثابت به چه چیزی فکر می‌کرده‌اید.

خودتان را جای خواننده بگذارید

یک تکنیک کلی که در این کتاب استفاده می‌کنیم این است که «تصور کنید کد شما در نظر یک شخص خارجی چگونه به نظر می‌رسد»، کسی که به اندازه شما با این پروژه آشنا نیست. این تکنیک به ویژه برای تشخیص اینکه چه بخش‌هایی نیاز به کامنت گذاری دارند به شما کمک می‌کند.

پیش بینی سوالات احتمالی



آیا کسی سوالی دارد؟...
که با تابلوهای نصب شده پوشش داده نشده باشد؟...

هنگامی که شخص دیگری کد شما را بخواند، بخش‌هایی وجود دارد که ممکن است در مورد بخش‌هایی از آن به این صورت فکر کند که «ها!!!!!!؟! اینا اصلاً در مورد چی هستند؟» کار شما این

هست که این بخش‌ها را کامنت گذاری کنید. به عنوان مثال نگاهی به تعریف تابع `Clear()` بیندازید:

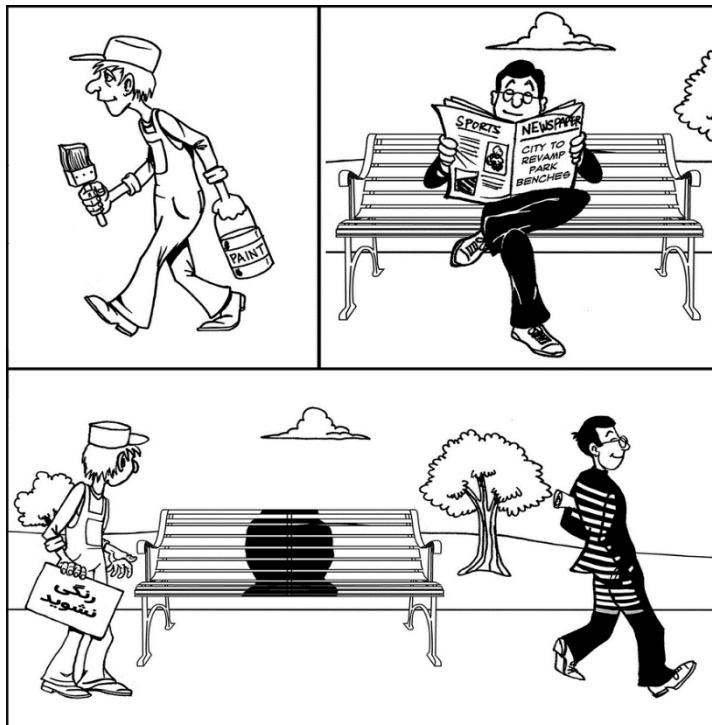
```
struct Recorder {
    vector<float> data;
    ...
    void Clear() {
        vector<float>().swap(data); // Huh? Why not just data.clear()?
    }
};
```

اکثر برنامه‌نویسان C++ در هنگام مواجه با این کد به این فکر خواهد کرد که: چرا در آن به جای تعویض^۱ با یک `vector` خالی، فقط از `data.clear()` استفاده نشده است؟ خب معلوم است که این تنها روش مجبور کردن یک `vector` است، تا به درستی حافظه خود را به حافظه^۲ `allocator` رها سازی کند. این کار از جزئیات C++ است که به خوبی شناخته نشده است. بنابراین، باید اینگونه کامنت گذاری شود:

```
// Force vector to relinquish its memory (Look up "STL swap trick")
vector<float>().swap(data);
```

^۱ swapping

^۲ allocators are a component of the C++ Standard Library.

اعلام کردن تله^۱ احتمالی

هنگام مستند کردن یک تابع یا یک کلاس، سوال خوبی که می‌توانید از خود بپرسید این است که، چه چیز تعجب آوری در مورد این کد وجود دارد؟ چگونه ممکن است این کد سبب برداشت اشتباه خواننده شود؟ در واقع، شما دارید «به آینده فکر می‌کنید» تا مشکلاتی که احتمالاً افراد در زمان استفاده از کد، به آن دچار می‌شوند را پیش بینی کنید.

به عنوان مثال، فرض کنید تابعی نوشته‌اید که یک ایمیل را به کاربری معین ارسال می‌کند:

```
void SendEmail(string to, string subject, string body);
```

پیاده سازی این تابع شامل اتصال به یک سرویس ایمیل خارجی است که ممکن است یک ثانیه كامل یا بیشتر زمان ببرد. شخصی که در حال نوشتتن یک برنامه کاربردی وب^۲ است ممکن است

^۱ Pitfall

^۲ web application

متوجه این موضوع نشده و اشتباها این تابع را حین انجام درخواست^۱ HTTP فراخوانی کند که در صورت قطع سرویس ایمیل، انجام این کار سبب توقف^۲ برنامه کاربردی وب می‌گردد.

برای جلوگیری از این اشتباه احتمالی، باید در مورد جزئیات پیاده سازی شده کامنت گذاری کنید:

```
// Calls an external service to deliver email. (Times out after 1 minute.)
void SendEmail(string to, string subject, string body);
```

در اینجا مثال دیگری داریم، فرض کنید تابع `FixBrokenHtml()` را دارید که تلاش می‌کند کدهای HTML دارای اشکال را با افزودن تگ بسته شدن به انتهای آن‌ها، بازنویسی کند:

```
def FixBrokenHtml(html): ...
```

این تابع در غیر از مواردی که حین اجرای آن تگ‌های بدون همتا^۳ و تو در توی بسیار زیادی وجود داشته باشد، خیلی عالی عمل می‌کند ولی برای ورودی‌های بهم ریخته و کثیف HTML اجرای این تابع می‌تواند چند دقیقه طول بکشد.

به جای اینکه اجازه دهید کاربر خودش بعدها متوجه این موضوع شود، بهتر است که در یک کامنت و در همان ابتدای کار در مورد سرعت اجرای آن هشدار دهید:

```
// Runtime is O(number_tags * average_tag_depth), so watch out for badly nested inputs.
def FixBrokenHtml(html): ...
```

^۱ request

^۲ Hang

^۳ Unmatched

کامنت‌های روند کلی^۱



یکی از سخت‌ترین موارد برای عضو جدید تیم، فهمیدن روند کلی برنامه است. اینکه تعامل کلاس‌ها چگونه است، جریان داده در کل سیستم به چه صورت بوده و نقاط ورودی^۲ کجاها هستند.

طراح اصلی سیستم به دلیل درگیری زیاد با این موارد معمولاً فراموش می‌کند که درباره آن‌ها کامنت گذاری کند.

حال بباید این آزمایش فکری را در نظر بگیرید: شخصی به تازگی به تیم شما اضافه شده است، او جلوی شما می‌نشیند تا شما او را با کدپایه آشنا کنید.

شما در حال ارائه توضیحات در مورد کدپایه، ممکن است به فایل‌ها و کلاس‌های خاصی اشاره کنید و چیزی شبیه این جملات را بگویید:

^۱ "Big Picture" Comments

^۲ entry points

- این کد واسطه^۱ بین منطق بیزینس ما و دیتابیس است. هیچ کد اپلیکیشنی نباید از این کد مستقیماً استفاده کند.
- این کلاس به نظر پیچیده می‌رسد، اما این در واقع یک Cache هوشمند^۲ است و در مورد بقیه سیستم چیزی نمی‌داند.

پس از یک دقیقه مکالمه غیر جدی، عضو جدید تیم نسبت به زمانی که خودش به تنها یکی می‌خواست سورس کد را بخواند، اطلاعات خوبی در مورد سیستم بدست می‌آورد.

این دقیقاً همان نوع اطلاعاتی است که باید به کامنت‌های سطح بالا^۳ اضافه شود.

در اینجا یک مثال ساده از کامنت سطح-فایل^۴ داریم:

```
// This file contains helper functions that provide a more convenient interface to our
// file system. It handles file permissions and other nitty-gritty details.
```

از این فکر که حتماً مجبور هستید یک سند رسمی و وسیع بنویسید نگران نشوید. انتخاب چند جمله خوب، بهتر از این است که هیچ چیزی نباشد.

کامنت‌های خلاصه

این ایده خوبی است که حتی در عمق یک تابع، در مورد تصویر کلی‌تر آن، کامنت گذاری کنید. در اینجا مثالی از یک کامنت آورده شده است که کد سطح-پایین^۵ زیر آن را به صورت خلاصه بیان می‌کند:

```
# Find all the items that customers purchased for themselves.
for customer_id in all_customers:
    for sale in all_sales[customer_id].sales:
        if sale.recipient == customer_id:
            ...

```

بدون این کامنت، خواندن هر خط از کد مقداری رمزآلود خواهد شد چرا که به نظر می‌رسد از طریق ... در حال تکرار هستیم و این سوال پیش می‌آید که «این تکرار برای چیست؟».

^۱ glue code

^۲ smart cache

^۳ high-level

^۴ file-level

^۵ low-level code

این کار زمانی مفید است که کامنتهای خلاصه را در یک تابع بزرگتر داشته باشیم، جایی که تعدادی از تکههای^۱ بزرگ کد داخل آنها وجود دارد.

```
def GenerateUserReport():
    # Acquire a lock for this user
    ...
    # Read user's info from the database
    ...
    # Write info to a file
    ...
    # Release the lock for this user
```

همچنین این کامنتها می‌توانند به عنوان گزارش‌های خلاصه از اینکه توابع چه کاری انجام می‌دهند، عمل کنند، بنابراین قبل از اینکه خوانند وارد جزئیات تابع شود، می‌تواند به طور خلاصه کار انجام گرفته توسط تابع را بفهمد(البته اگر این تکه‌ها به آسانی قابل جدا شدن از یکدیگر هستند، بهتر است که آنها را به شکل توابع جداگانه بنویسیم. همان‌گونه که قبلاً اشاره کردیم، کد خوب بهتر از کد بد با کامنت خوب است).

دلایل کامنت گذاری: برای چه چیزی؟ چرا؟ یا چگونه؟

ممکن است توصیه‌های سختگیرانه‌ای از این دست را شنیده باشید که: کامنت گذاری باید برای «چرا باید انجام شود» باشد و نه برای «چه چیزی یا چگونه». این توصیه اگرچه قابل توجه است ولی ما فکر می‌کنیم که این دستورات خیلی ساده هستند و برای افراد مختلف معنای متفاوتی می‌دهند. توصیه ما این است که هر کاری که به خواننده برای درک ساده‌تر کد کمک می‌کند را انجام دهید. این امر ممکن است شامل کامنت گذاری در مورد چه چیزی، چگونه، یا چرا و یا حتی شامل هر سه این موارد باشد.

افکار نهایی، گذر از بلوك نویسنده^۲

بسیاری از برنامه‌نویسان کامنت نویسی را دوست ندارند چرا که احساس می‌کنند باید کار زیادی برای نوشتتن یک کامنت خوب، انجام شود. بهترین راه حل در مواجهه با این مانع احساسی این است که شروع به نوشتتن کنید. بنابراین دفعه بعدی که در مورد نوشتتن کامنت تردید داشتید، فقط جلو رفته و کامنتی در این مورد این که به چه چیزی فکر می‌کنید بنویسید، هرچند که کامل نباشد.

^۱ chunks

^۲ Getting Over Writer's Block

به عنوان مثال فرض کنید شما روی یک تابع کار می‌کنید و با خود می‌اندیشید که: **لعنت بهش!**، اگر تکراری در این لیست وجود داشته باشد، این چیزها^۲ دچار مشکل خواهد شد. فقط همین را به عنوان کامنت بنویسید:

```
// Oh crap, this stuff will get tricky if there are ever duplicates in this list.
```

دیدید! خیلی سخت نبود. در واقع کامنت بدی هم نبود، هرچند کمی مبهم است ولی مطمئناً بهتر از هیچ است. برای حل این مشکل نیز می‌توانیم هر عبارت را مرور کرده و یک چیز خاص‌تر را جایگزین آن کنیم:

- کلمه‌ای با معنی واقعی‌تر برای **لعنت بهش!** همچون این عبارت: مراقب باشید: این چیزی است که باید بیشتر مراقب آن باشید.
- کلمه‌ای با معنی دقیق‌تر نیز برای **this stuff** همچون این عبارت: کدی که این ورودی را مدیریت می‌کند.
- عبارتی رساتر برای اجرای آن دشوار^۳ خواهد بود همچون این عبارت: پیاده سازی آن سخت خواهد بود.

احتمالاً کامنت جدید به این صورت خواهد بود:

```
// Careful: this code doesn't handle duplicates in the list (because that's hard to do)
```

- توجه داشته باشید که ما نوشتن یک کامنت را به سه مرحله ساده تقسیم کردیم:
۵. هرچیزی که به عنوان کامنت در مغز شما است را بنویسید.
 ۶. کامنت را بخوانید، و هر چیزی را که نیازمند بهبود است، بهبود دهید.
 ۷. کل کامنت را بهبود دهید.

هر چه کامنت بیشتری بنویسید، متوجه خواهید شد که کیفیت کامنتها در هر مرحله بهتر و بهتر می‌شوند و در نهایت ممکن است کامنت شما اصلاً نیازی به اصلاح نداشته باشد. همچنین با کامنت گذاری در اوایل و اغلب اوقات، از وضعیت ناخوشایینِ نیاز به نوشتن یک دسته از کامنتها در پایان، خودداری خواهید کرد.

^۱ Oh crap

^۲ this stuff

^۳ will get tricky

خلاصه فصل

هدف از یک کامنت، کمک به خواننده است تا بداند نویسنده کد هنگام نوشتن کدنویسی چه چیزی می‌داند. کل این فصل درباره فهمیدن همه قطعه اطلاعات نه چندان واضحی است که درباره کد در ذهن خود دارید و آن‌ها را بشکل کامنت می‌نویسید.

چه چیزی را نباید کامنت کنیم:

- واقعیت‌هایی که به سرعت می‌توان از خود کد به دست آورد.
- کامنت‌های کمکی^۱ که برای کد بد ایجاد شده است، مانند نام‌گذاری بد برای یک تابع، که به جای آن کد را بهبود دهید.

افکاری که باید ثبت کنید شامل این موارد است:

- بیشن در مورد اینکه چرا کد به این شیوه است و نه شیوه دیگر(تفسیر کارگردان^۲)
- نمایش کاستی‌های کد با استفاده از نشانگرهای مانند TODO: یا XXXX:
- توضیحی در این مورد که مقدار یک ثابت از کجا به دست آمده است.

خود را جای خواننده قرار دهید:

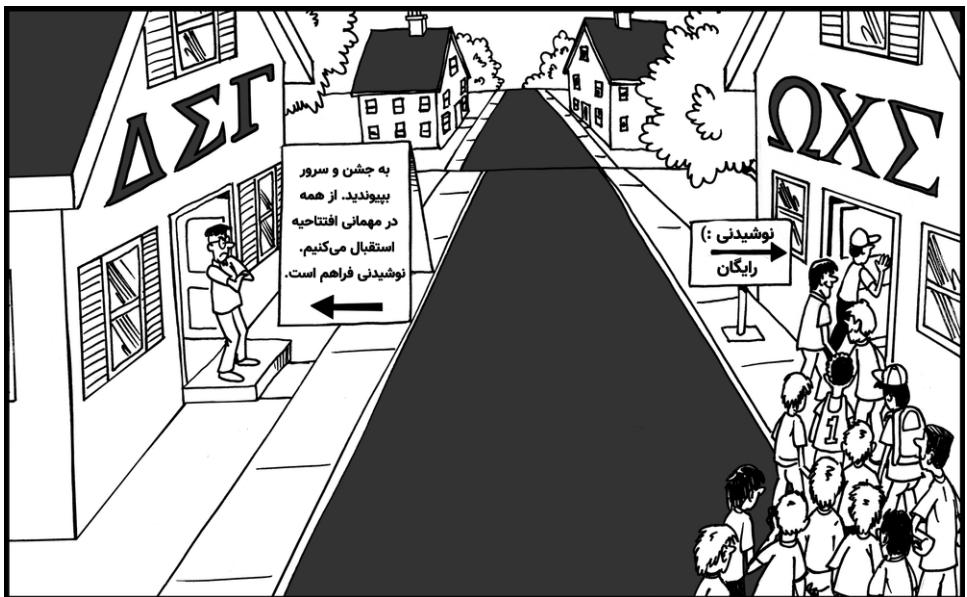
- بیش بینی کنید که کدام بخش از کد شما باعث می‌شود تا خواننده بگوید: ها!؟!؟!؟ و برای آن کامنت بنویسید.
 - رفتارهای تعجب برانگیز را که یک خواننده متوسط انتظارش را ندارد، مستند کنید.
 - از کامنت گذاری تصویر کلی در سطح فایل یا کلاس، برای شرح چگونگی قرار گیری همه این‌ها در کنار هم استفاده کنید.
- بلوک‌های کد را توسط کامنت خلاصه کنید تا خواننده در جزئیات گم نشود.

^۱ Crutch comments

^۲ director commentary

فصل ششم

ساختن کامنت‌ها به شکل دقیق و خلاصه



موضوع فصل قبل این بود که بفهمیم «چه چیزی را باید کامنت گذاری کنیم»، این فصل در این مورد است که «چگونه کامنتی بنویسیم که دقیق و خلاصه باشد».

اگر می‌خواهید یک کامنت خوب بنویسید، باید دقیق، خاص و در حد امکان با جزئیات باشد. اما از آنجا که، کامنتها فضای اضافی روی صفحه نمایش را اشغال کرده و زمان بیشتری را برای خواندن کد می‌گیرند باید مختصر(خلاصه و جمع و جور) باشند.

کلید طلایی

کامنتها باید نسبت به فضایی که اشغال می‌کنند، نزد بالایی از اطلاعات را شامل شوند.

در ادامه این فصل مثال‌هایی داریم که نشان می‌دهد چگونه این کار را انجام دهید.

کامنتها را خلاصه نگه دارید

در اینجا مثالی از یک کامنت برای تعریف یک `DataType` در C++ را ملاحظه می‌کنید:

```
// The int is the CategoryType.
// The first float in the inner pair is the 'score',
// the second is the 'weight'.
typedef hash_map<int, pair<float, float> > ScoreMap;
```

اما چرا یک کامنت سه خطی برای توضیح آن استفاده شده است، وقتی که شما می‌توانید آن را فقط در یک خط نشان دهید؟

```
// CategoryType -> (score, weight)
typedef hash_map<int, pair<float, float> > ScoreMap;
```

هرچند بعضی کامنتها به سه خط از فضا نیاز دارند، اما بی‌شك این یکی از آن مدل‌ها نیست.

از استفاده از ضمایر مبهم خودداری کنید

همان‌گونه که می‌دانید در تلفظ زبان انگلیسی ممکن است جمله Who's on First معانی مختلفی داشته باشد و می‌تواند خواننده را دچار ابهام کند چرا که خواننده باید با انجام کار اضافی مشکل مرجع ضمیر این جمله را حل کند. در بعضی موارد، این که کلمات `it` یا `this` به چه چیزی اشاره می‌کنند، واضح نیست. به عنوان مثال:

```
// Insert the data into the cache, but check if it's too big first.
```

^۱ information-to-space

در این کامنٹ، کلمه `it` ممکن است به داده یا حافظه موقت اشاره کند. هرچند شما با خواندن بقیه کد ممکن است بفهمید که دقیقاً به چه چیزی اشاره می‌کند ولی اگر مجبور شوید که این کار را انجام دهید، پس چه مزیتی در استفاده از کامنٹ وجود دارد؟

در صورت وجود احتمال هرگونه ابهام یا سردرگمی، بهترین کار این است که به جای ضمیرهایی مانند `it` و `this`، کلمه اصلی جایگزین شود. فرض کنید در مثال قبلی منظور از `it` کلمه `the data` است:

```
// Insert the data into the cache, but check if the data is too big first.
```

این ساده‌ترین تغییر برای درست کردن آن است. همچنین می‌توانید با بازنویسی جمله کلمه `it` را واضح‌تر کنید:

```
// If the data is small enough, insert it into the cache.
```

جملات درهم و برهم لهستانی^۱

در بسیاری از موارد دقیق‌تر کردن یک کامنٹ وابسته به خلاصه کردن آن است. در اینجا مثالی از یک خزنه وب^۲ داریم:

```
# Depending on whether we've already crawled this URL before, give it a different priority.
```

شاید به نظر برسد که این عبارت جمله خوبی است، اما آن را با کامنٹ زیر مقایسه کنید:

```
# Give higher priority to URLs we've never crawled before.
```

بی‌شک این یکی ساده‌تر، کوتاه‌تر و دقیق‌تر است. همچنین توضیح می‌دهد که اولویت بالاتر^۳ به URL‌های کشف نشده اختصاص داده شده است در حالی که کامنٹ قبلی حاوی این اطلاعات نبود.

رفتار یکتابع را به صورت دقیق شرح دهید

تصور کنید شما فقط یک تابع نوشته‌اید که تعداد خطوط یک فایل را می‌شمارد:

```
// Return the number of lines in this file.
```

```
int CountLines(string filename) { ... }
```

^۱ Polish Sloppy Sentences

^۲ web crawler

^۳ higher

این کامنت خیلی دقیق نیست. روش‌های خیلی زیادی برای تعریف یک خط یا line وجود دارد. در اینجا چند مورد corner^۱ را برای فکر کردن به آن آورده‌ایم:

- "" (an empty file)—0 or 1 line?
- "hello"—0 or 1 line?
- "hello\n"—1 or 2 lines?
- "hello\n world"—1 or 2 lines?
- "hello\n\r cruel\n world\r"—2, 3, or 4 lines?

ساده‌ترین پیاده سازی این است که کاراکترهای خط جدید(\n) را بشماریم. (این همان روشی است که سیستم عامل Unix در دستور wc با آن کار می‌کند). در ادامه کامنتی بهتر، منطبق با این پیاده سازی را مشاهده می‌کنید:

```
// Count how many newline bytes ('\n') are in the file.
int CountLines(string filename) { ... }
```

با اینکه این کامنت خیلی طولانی‌تر از نسخه اول نیست، اما شامل اطلاعات بیشتری است. این کامنت به خواننده می‌گوید که، اگر هیچ خط جدیدی وجود نداشته باشد، تابع مقدار 0 را بر می‌گرداند.^۲ همچنین می‌گوید که کاراکتر سرخط(r) نادیده گرفته شده است.

از مثال‌های ورودی/خروجی^۳ که موردهای Corner را نشان می‌دهد استفاده کنید

وقتی صحبت از کامنت می‌شود، یک مثال ورودی/خروجی که با دقت انتخاب شده باشد، می‌تواند به اندازه هزاران کلمه ارزش داشته باشد.

به عنوان مثال در اینجا یک تابع معمولی داریم که بخشی از یک رشته را حذف می‌کند:

```
// Remove the suffix/prefix of 'chars' from the input 'src'.
String Strip(String src, String chars) { ... }
```

این کامنت خیلی دقیق نیست زیرا نمی‌تواند به این سوالات پاسخ دهد:

^۱ در مهندسی، یک corner case شامل یک مشکل یا وضعیتی است که فقط در خارج از پارامترهای عملیاتی معمولی اتفاق می‌افتد.

^۲ return

^۳ Carriage return

^۴ Input/output

- آیا `chars` یک زیر رشته است که باید حذف شود و یا فقط مجموعه‌ای از متن‌های نامرتب است؟
- اگر چندین `chars` در پایان `src` وجود داشته باشد چه می‌شود؟
- به جای آن، یک مثال خوب که می‌تواند به این سوالات پاسخ دهد عبارت است از:

```
// ...
// Example: Strip("abba/a/ba", "ab") returns "/a/"
String Strip(String src, String chars) { ... }
```

این مثال عملکرد `Strip()` را به شکل کامل نشان می‌دهد. توجه داشته باشید که یک مثال ساده‌تر، اگر به سوالات پاسخ ندهد، مفید نخواهد بود:

```
// Example: Strip("ab", "a") returns "b"
```

در اینجا مثال دیگری از تابعی که می‌تواند به طور مشابه استفاده شود، داریم:

```
// Rearrange 'v' so that elements < pivot come before those >= pivot;
// Then return the largest 'i' for which v[i] < pivot (or -1 if none are < pivot)
int Partition(vector<int>* v, int pivot);
```

هر چند این کامنت خیلی دقیق است، اما تصور آن کمی دشوار می‌باشد. در ادامه مثالی داریم که می‌توانید برای نشان دادن اطلاعات دقیق‌تر، موارد بیشتری را به کامنت اضافه کنید:

```
// ...
// Example: Partition([8 5 9 8 2], 8) might result in [5 2 | 8 9 8] and return 1
int Partition(vector<int>* v, int pivot);
```

- نکاتی در مورد مثال‌های خاصی که برای ورودی/خروجی انتخاب کردیم وجود دارد که عبارتند از:
- مقدار `pivot` برابر است با المانی در وکتور برای نشان دادن مورد لبها^۱.
 - ما کپی‌هایی را داخل `vector(8)` قرار دادیم برای اینکه نشان دهیم ورودی قابل قبول است.

^۱ یک مورد لبها¹ (edge case) یک مشکل یا وضعیتی است که تنها به ازای مقادیر حداقل یا حداقل یک پارامتر رخ می‌دهد.

- نتیجه vector مرتب شده نیست. در صورت مرتب شده بودن ممکن بود خواننده ایده اشتباхи از آن برداشت کند.
- از آنجا که مقدار برگشته برابر ۱ بود، اطمینان حاصل کردیم که دیگر ۱ به عنوان یک مقدار داخل وکتور نباشد چرا که گیج کننده می‌شد.

هدف کد خود را مشخص کنید

همانگونه که در فصل قبل اشاره کردیم، اغلب کامنت گذاری درباره این است که به خواننده بگویید شما هنگام نوشتن کد در حال فکر کردن به چه چیزی بوده‌اید. متاسفانه بسیاری از کامنت‌ها بدون اینکه اطلاعات جدیدی ارائه دهند، فقط توصیف می‌کنند که کد چه کاری انجام می‌دهد. در اینجا مثالی از این نوع کامنت داریم:

```
void DisplayProducts(list<Product> products) {
    products.sort(CompareProductByPrice);
    // Iterate through the List in reverse order
    for (list<Product>::reverse_iterator it = products.rbegin(); it != products.rend();
        ++it)
        DisplayPrice(it->price);
    ...
}
```

تمام کاری که این کامنت انجام می‌دهد این است که فقط خط پایین خود را توصیف می‌کند. به جای آن این کامنت بهتر را در نظر بگیرید:

```
// Display each price, from highest to lowest
for (list<Product>::reverse_iterator it = products.rbegin(); ... )
```

این کامنت توضیح می‌دهد که برنامه در سطح بالا در حال انجام کار بوده و این با آنچه برنامه‌نویس هنگام نوشتن کد به آن فکر می‌کرد، خیلی بیشتر منطبق است.

خصوصاً که در این برنامه یک اشکال^۱ وجود دارد! تابع CompareProductByPrice (نمایش داده نشده) ابتدا آیتم‌های با قیمت بالاتر را مرتب می‌کند و این برعکس آن چیزی است که نویسنده قصد انجامش را داشته است.

^۱ bug

این دلیل خوبی برای بهتر بودن کامنت دوم است. با وجود این اشکال، کامنت اول از نظر فنی درست است (حلقه به ترتیب معکوس تکرار می‌شود). اما با کامنت دوم، احتمالاً خواننده بهتر متوجه می‌شود که هدف نویسنده ابتدا نشان دادن آیتم‌ها با قیمت بالاتر بوده، که این دقیقاً کاری که واقعاً کد انجام می‌دهد را در کامنت تکرار نکرده است.

در واقع این کامنت به عنوان یک بررسی افزونگی^۱ عمل می‌کند.

هر چند در نهایت بهترین روش بررسی افزونگی استفاده از تست واحد است (در فصل ۱۴ به آن پرداخته شده است). اما هنوز هم کامنت‌هایی مانند این، که در مورد هدف برنامه شما توضیح می‌دهند، ارزشمند هستند.

کامنت گذاری نام پارامترهای تابع

فرض کنید یک فراخوانی تابع شبیه کد زیر را مشاهده می‌کنید:

```
Connect(10, false);
```

این فراخوانی تابع به دلیل اینکه integer و Boolean به آن پاس می‌شود، کمی مبهم است.

در زبان‌های شبیه پایتون، شما می‌توانید نامی را به آرگومان‌ها اختصاص دهید:

```
def Connect(timeout, use_encryption): ...  
# Call the function using named parameters  
Connect(timeout = 10, use_encryption = False)
```

اما در زبان‌های شبیه C++ و Java که امکان چنین کاری نیست، می‌توانید از کامنت inline به همین منظور استفاده نمایید:

```
void Connect(int timeout, bool use_encryption) { ... }  
// Call the function with commented parameters  
Connect(/* timeout_ms = */ 10, /* use_encryption = */ false);
```

توجه داشته باشید که پارامتر اول را به جای timeout_ms با عبارت timeout_۱۰ نام‌گذاری کردیم. در حالت ایده‌آل، آرگومان واقعی تابع timeout_ms خواهد بود، اما اگر به دلایلی نتوانیم این

^۱ redundancy check

تغییر(یعنی اضافه کردن `ms`_ به انتهای نام در کامنت) را انجام دهیم، این یک روش دستی برای بهبود نام خواهد بود.

وقتی آرگومان‌های Boolean استفاده می‌شوند، این خیلی مهم است که در جلوی مقدار، کامنت `/* = name */` را قرار دهید. قرار دادن کامنت بعد از مقدار خیلی گیج کننده است:

```
// Don't do this!
Connect( ... , false /* use_encryption */);
// Don't do this either!
Connect( ... , false /* = use_encryption */);
```

در این مثال‌ها، اینکه معنی کلمه `false` جمله `use encryption` است یا `don't use encryption` واضح نیست.

هر چند اکثر فراخوانی توابع، نیازی به کامنت‌هایی شبیه این ندارند، اما این روش دستی(و جمع و جور) برای توضیح آرگومان‌هایی که مبهم به نظر می‌رسند، بسیار مفید است.

از کلمات **Information-Dense** استفاده کنید

بعد از چندین سال برنامه‌نویسی، متوجه می‌شوید که مشکلات و راه حل‌های مشابه به صورت مرتب پیش می‌آید. اغلب، کلمات یا اصطلاحات خاصی که برای توصیف این الگوهای اصطلاحات ایجاد شده‌اند، وجود دارد. استفاده از این کلمات می‌تواند کامنت‌های شما را بسیار جمع و جورتر کند.

به عنوان مثال، فرض کنید شما چنین کامنتی داشتید:

```
// This class contains a number of members that store the same information as in the
// database, but are stored here for speed. When this class is read from later, those
// members are checked first to see if they exist, and if so are returned; otherwise the
// database is read from and that data stored in those fields for next time.
```

به جای آن شما می‌توانستید فقط بگویید:

```
// This class acts as a caching layer to the database.
```

¹ value

همچنین به عنوان مثالی دیگر کامنتی شبیه این را در نظر بگیرید:

```
// Remove excess whitespace from the street address, and do lots of other cleanup
// like turn "Avenue" into "Ave." This way, if there are two different street addresses
// that are typed in slightly differently, they will have the same cleaned-up version and
// we can detect that these are equal.
```

به جای آن می‌توانستید بگویید:

```
// Canonicalize the street address (remove extra spaces, "Avenue" -> "Ave.", etc.)
```

کلمات و اصطلاحات زیادی وجود دارد که معانی خیلی زیادی را در خود دارند، مانند کلمه heuristic، naive solution و کلمات مشابه دیگر. اگر کامنتی دارید که احساس می‌کنید کمی طولانی است، بررسی کنید که آیا می‌توان آن را به عنوان یک وضعیت برنامه‌نویسی معمولی توصیف کرد یا خیر؟

خلاصه فصل

این فصل درباره نوشتن کامنت‌هایی است که اطلاعات زیادی را در یک فضای کوچک و در کمترین زمان ممکن در اختیار شما قرار می‌دهد. نکات خاص این فصل عباتند از:

- از ضمیرهایی مانند it و this (هنگامی که به چندین چیز اشاره می‌کنند) خودداری کنید.
- رفتار یک تابع را با همان دقت عملی توصیف کنید.
- کامنت‌های خود را با مثال‌های ورودی/خروجی که با دقت انتخاب شده‌اند، نشان دهید.
- به جای جزئیات، هدف سطح بالای کد خود را واضح بیان کنید.
- از کامنت‌های درون خطی یا inline (مثلاً `Function(* arg = ...)`) برای توضیح ابهام آرگومان‌های تابع استفاده کنید.
- کامنت‌های خود را با استفاده از کلماتی که معنای زیادی را در خود دارند کوتاه نگه دارید.

بخش دوم

ساده سازی حلقه‌های تکرار و منطق^۱

ما در بخش اول بهبودهای سطح-ظاهری را پوشش دادیم. راههایی ساده برای بهبود خوانایی کد شما به صورت خط به خط که می‌توانند بدون ریسک یا تلاش زیادی انجام شوند. در بخش بعدی، ما عمیق‌تر می‌شویم و درباره منطق و تکرارهای برنامه بحث می‌کنیم: کنترل جریان، عبارات منطقی^۲ و متغیرهایی که باعث می‌شوند کد شما کار کند. مثل همیشه، هدف ما این است که این بخش از کد شما نیز ساده و قابل فهم شوند.

کلید اصلی این بخش تلاش برای حداقل^۳ کردن تعداد چمدان‌های ذهنی^۴ یک کد است. هر وقت شما یک حلقة پیچیده، یک اصطلاح غول پیکر یا تعداد زیادی از متغیرها را دیدید، این‌ها به چمدان ذهنی موجود در سر شما اضافه شده و سبب می‌شوند که سخت‌تر فکر کنید و دفعات بیشتری به یاد آورید که این امر دقیقاً برعکس «درک آسان» است. زمانی که کد دارای تعداد زیادی چمدان ذهنی است، به احتمال زیاد متوجه اشکالات نشده و تغییر دادن کد سخت‌تر شده و کار با آن لذت کمتری خواهد داشت.

^۱ Loops and Logic

^۲ logical expressions

^۳ minimize

^۴ mental baggage

فصل هفتم

ساده کردن خوانایی کنترل جریان



اگر در یک کد هیچ شرط^۱، حلقه^۲ یا دیگر دستورات کنترل جریان وجود نداشته باشد، خواندش بسیار ساده خواهد بود. این نوع پرش‌ها و شاخه‌ها چیزهای سختی هستند که می‌توانند کد را سریعاً مبهم کنند. این فصل درباره ایجاد سادگی در خواندن کنترل جریان، در کدهای شما است.

کلید طلایی

تمام شرطها، حلقه‌ها و سایر تغییرات برای کنترل جریان را تا حد امکان به شکل طبیعی^۳ انجام دهید. به شکلی بنویسید که خواننده مجبور نشود توقف کند و مجدداً کد شما را بخواند.

آرگومان‌ها را در شرط‌ها مرتب کنید

کدام یک از این دو قطعه کد خوانایی بیشتری دارند، این کد:

```
if (length >= 10)
```

یا این:

```
if (10 <= length)
```

برای اکثر برنامه‌نویسان، کد اول خوانایی بیشتری دارد. حال قضاوت شما درباره دو کد زیر چیست؟ مثلاً این:

```
while (bytes_received < bytes_expected)
```

یا این:

```
while (bytes_expected > bytes_received)
```

در اینجا هم، نسخه اول خوانایی بیشتری دارد. اما چرا؟ قانون کلی چیست؟ چگونه تصمیم می‌گیرید که بهتر است کد را به صورت $b < a$ بنویسد و یا به صورت $a > b$ ؟

یک راهنمایی مفید به نظر ما این است:

سمت چپ	سمت راست
عبارتی که مقدارش در حال تغییر کردن است را در سمت چپ بنویسید.	عبارتی که مقدار آن معمولاً ثابت است را در سمت راست بنویسید.

^۱ conditionals

^۲ loops

^۳ natural

این راهنمای با قواعد زبان انگلیسی مطابقت دارد. کاملاً طبیعی است که بگوییم «اگر حداقل ۱۰۰ هزار دلار در سال به دست آورید^۱» یا «اگر حداقل ۱۸ سال دارید^۲». و نه اینکه بگوییم «اگر ۱۸ سال کمتر یا مساوی با سن شما است^۳».

این مثال به خوبی خواناتر بودن while (bytes_received < bytes_expected) را توضیح می‌دهد، چرا که bytes_received مقداری است که بررسی روی آن انجام می‌شود و با اجرای حلقه، مقدار آن در حال افزایش بوده و bytes_expected مقدار پایدارتری^۴ است که در برابر آن مورد مقایسه قرار می‌گیرد.

پیغام YODA: آیا هنوز هم مفید است؟

در برخی از زبان‌ها از جمله C و C++, و نه Java عمل انتصاف در یک شرط if قانونی است:

```
if (obj = NULL) ...
```

به احتمال زیاد این کد دارای اشکال است و منظور اصلی برنامه‌نویس این بوده است:

```
if (obj == NULL) ...
```

برای جلوگیری از چنین اشکالاتی، اکثر برنامه‌نویسان ترتیب آرگومان‌ها را با هم جابجا می‌کنند:

```
if (NULL == obj) ...
```

در این روش == if به طور تصادفی به شکل = نوشته می‌شود، ولی مشکل این است که عبارت if(NULL = Obj) حتی کامپایل هم نمی‌شود.

متاسفانه، جابجایی ترتیب آرگومان‌ها سبب می‌شود که برای خواندن کمی غیرطبیعی جلوه کند. خوشبختانه کامپایلرهای مدرن نسبت به کدهایی مانند if(NULL = Obj) هشدار می‌دهند.

^۱ if you make at least \$100K/year

^۲ if you are at least 18 years old

^۳ if 18 years is less than or equal to your age.

^۴ stable

ترتیب بلوک‌های if/else



هنگام نوشتن یک دستور if/else، به طور معمول باید آزادی عمل در مورد جابجا کردن ترتیب بلوک‌های دستور را داشته باشید. به عنوان نمونه شما می‌توانید به هر دو شکل زیر شرط خود را بنویسید:

```
if (a == b) {
    // Case One ...
} else {
    // Case Two ...
}
```

و یا:

```
if (a != b) {
    // Case Two ...
} else {
    // Case One ...
}
```

ممکن است تا کنون به این موضوع فکر نکرده باشید، اما در بعضی از موارد دلایل خوبی برای ترجیح دادن یک ترتیب نسبت به دیگری وجود دارد:

- ترجیح دادن بررسی مورد مثبت، به جای مورد منفی در ابتدای کار. به عنوان مثال ترجیح عبارت `if(!debug)` نسبت به `.if(debug)`.
 - ترجیح دهید بررسی مورد ساده‌تر در ابتدای کار انجام شود تا زودتر از این شرط خارج شوید. همچنان این رویکرد ممکن است اجازه دهد `if` و `else` با هم و در یک زمان واحد، روی صفحه نمایشگر قابل دیدن باشند، که خوب است.
 - ترجیح دادن بررسی مورد جالب‌تر یا انگشت نماتر^۱ در ابتدای کار.
 - گاهی این ترجیح دادن‌ها با هم تداخل داشته و شما باید در مورد آن‌ها تصمیم گیری کنید. اما در بسیاری از موارد، مشخص است که کدام گزینه برزنه است.
- به عنوان مثال فرض کنید یک وب سرور دارید که هر گاه URL شامل پارامتر کوئری `expand_all` بود، پاسخی ایجاد می‌کند:

```
if (!url.HasQueryParameter("expand_all")) {
    response.Render(items);
    ...
} else {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    ...
}
```

وقتی که خواننده به خط اول نگاه می‌کند، مغز او بلاfacسله به `expand_all` فکر می‌کند. همچون زمانی که کسی می‌گوید: «به یک فیل صورتی فکر نکن». کاری از شما ساخته نیست، بی شک در مورد آن فکر خواهید کرد. کلمه غیر معمول فیل صورتی باعث می‌شود تا عبارت فکر نکن به چشم نیامده و شخص به فیل صورتی فکر کند.

^۱ conspicuous

در اینجا `expand_all` همان فیل صورتی است. از آنجا که مورد جالب‌تری بوده و هم زمان نیز مثبت است، اجازه دهید آن را در ابتدا بیان کنیم:

```
if (url.HasQueryParameter("expand_all")) {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    ...
} else {
    response.Render(items);
    ...
}
```

از سوی دیگر، مثال زیر وضعیتی است که در آن مورد منفی ساده‌تر و جالب‌تر یا خطرناک‌تر بوده و به همین دلیل، ابتدا آن را مطرح می‌کنیم:

```
if not file:
    # Log the error ...
else:
    # ...
```

همواره این کار یک قضاوت یا تصمیم‌گیری است که به جزئیات هر مورد بستگی دارد. به طور خلاصه، توصیه ما این است که صرفاً به این فاکتورها توجه کنید و مراقب مواردی که if/else شما در یک ترتیب ناخوشایند قرار گرفته اند، باشید.

عبارت شرطی : (”a.k.a. “Ternary Operator”)

در زبان‌هایی مانند C شما می‌توانید یک عبارت شرطی را به صورت `cond ? a : b` بنویسید که روشی خاص و خلاصه برای نوشتمن عبارت `{ a } else { b }` if (`cond`) است.

البته تاثیر آن بر روی خوانایی قابل بحث است ولی طرفداران آن فکر می‌کنند این روشی خوب برای نوشتمن چیزی در یک خط است که در غیر این صورت به چندین خط نیاز دارد. مخالفان آن نیز

استدلال می‌کنند که این روش می‌تواند برای خواننده گیج کننده باشد و بررسی مرحله‌ای آن در دیباگ^۱ نیز سخت خواهد بود.

در اینجا موردی وجود دارد که عملگر سه گانه در عین مختصر بودن به راحتی قابل خواندن است:

```
time_str += (hour >= 12) ? "pm" : "am";
```

در صورت اجتناب از عملگر سه گانه، احتمالاً چیزی شبیه کد زیر خواهد داشت که کمی طولانی شده و باعث افزونگی^۲ گشته است:

```
if (hour >= 12) {
    time_str += "pm";
} else {
    time_str += "am";
}
```

در مورد زیر، هر چند بیان عبارت شرطی، به نظر منطقی‌تر می‌باشد، ولی با این حال، این عبارت می‌تواند خواندن سریع کد در یک نگاه را سخت کند:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

در اینجا، عملگر سه گانه تنها انتخاب بین دو مقدار ساده نیست. انگیزه نوشتن کدی شبیه این کد، این است که همه چیز در یک خط فشرده شود.

کلید طلایی

به جای حداقل کردن تعداد خطوط کد، معیار بهتر، حداقل کردن مدت زمانی است که یک نفر بتواند آن را درک کند.

بیان منطق با یک دستور if/else باعث می‌شود، کد طبیعی‌تر به نظر برسد:

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

^۱ debugger

^۲ redundant

توصیه

به طور پیش‌فرض همواره از if/else استفاده کنید. از حالت سه گانه؟ : فقط در صورتی استفاده کنید که مورد ساده‌ای باشد.

از حلقه‌های do/while اجتناب کنید

زبان Perl و بسیاری از زبان‌های برنامه‌نویسی قابل احترام، دستور زیر را در خود دارند:

`do { expression } while (condition)`

این عبارت یا حداقل یک بار اجرا می‌شود. به عنوان مثال کد زیر را در نظر بگیرید:

```
// Search through the List, starting at 'node', for the given 'name'.
// Don't consider more than 'max_length' nodes.

public boolean ListHasNode(Node node, String name, int max_length) {
    do {
        if (node.name().equals(name))
            return true;
        node = node.next();
    } while (node != null && --max_length > 0);
    return false;
}
```

آنچه در مورد حلقه do/while عجیب است این است که، یک بلوک از کد ممکن است بر اساس یک شرط در پایین آن کد مجدداً اجرا شود. به طور معمول شرط‌های منطقی در بالای کد بوده و نقش محافظ را دارند. این همان روشی است که دستورات if، for و while با آن کار می‌کنند. از آنجا که معمولاً کد را از بالا به پایین می‌خوانید این کار باعث می‌شود که حلقه do/while کمی غیرمعمول به نظر برسد. بسیاری از خوانندگان در نهایت مجبور می‌شوند کد را دوباره بخوانند.

این در حالی است که قاعده‌تا باید حلقه‌ها راحت‌تر خوانده شوند، زیرا شما قبل از اینکه بخشن داخلی بلوک کد را بخوانید، می‌دانید که شرط برای همه تکرارها چیست. ولی با این حال این احتمانه است که فقط برای حذف do/while کد دو برابر شود:

```
// Imitating a do/while – DON'T DO THIS!
body
while (condition) {
    body (again)
}
```

خوشبختانه ما متوجه شده‌ایم که با تمرین بیشتر می‌توان حلقه‌های do/while را به صورت حلقه‌های شروع شده با while نوشت:

```
public boolean ListHasNode(Node node, String name, int max_length) {
    while (node != null && max_length-- > 0) {
        if (node.name().equals(name)) return true;
        node = node.next();
    }
    return false;
}
```

مزیت دیگری که این نسخه دارد، این است که هنوز هم اگر node max_length برابر 0 و یا null باشد، هنوز هم کار می‌کند. دلیل دیگر برای اجتناب از do/while این است که دستور continue در داخل آن می‌تواند گیج کننده باشد. به عنوان مثال، به نظر شما این کد چه کاری انجام می‌دهد؟

```
do {
    continue;
} while (false);
```

این حلقه دائمی است یا فقط یک بار اجرا می‌شود؟ اکثر برنامه‌نویسان مجبور هستند که صبر کرده و درباره آن فکر کنند.

آقای Bjarne Stroustrup، سازنده C++ بهترین جمله را در زبان برنامه‌نویسی C++ درباره حلقه do/while اینگونه بیان می‌کند:

«تجربه به من ثابت کرده است که دستور do منبع خطاهای و گیج شدن‌ها است... من ترجیح می‌دهم شرط را در ابتدا ببینم، به همین دلیل ترجیح می‌دهم که از دستورات do استفاده نکنم.»

برخی از کدنویسان معتقد‌داند که توابع هیچ‌گاه نباید چندین دستور return داشته باشند. این عبارت بی معنی است. return زودهنگام از یک تابع، یک عملکرد کاملاً مناسب و در اغلب مواقع پسندیده است. به عنوان مثال این کد را ببینید:

```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (substr.equals("")) return true;
    ...
}
```

پیاده سازی این تابع بدون این بندهای شرطی^۱ خیلی غیر معمول خواهد بود.

یکی از انگیزه‌ها برای داشتن یک نقطه خروج از تابع، این است که تضمین می‌شود کلیه کدهای تمیز شده (cleanup)^۲ در بخش پایین تابع فراخوانی شده باشند. اما زبان‌های مدرن روش‌های پیچیده‌تری برای دستیابی به این ضمانت را ارائه می‌دهند:

زبان برنامه‌نویسی	اصطلاحات ساختاری برای cleanup
C++	destructors
Java, Python	try finally
Python	with
C#	using

¹ guard clauses

² اصطلاح Clean Up در برنامه نویسی به این معنی است که مطمئن شوید همه منابع (مانند فایل‌های باز شده، ارتباطات دیتابیس‌ها و غیره) که اخیراً استفاده شده‌اند به درستی بسته شوند، تا از نشت اطلاعات و منابع جلوگیری شود.

در زبان خالص C هنگام خروج یک تابع، مکانیزمی برای اعمال^۱ کد خاص وجود ندارد. بنابراین اگر یک تابع بزرگ با تعداد زیادی از کدهای cleanup وجود داشته باشد، return کردن سریع آن‌ها به طور صحیح احتمالاً سخت خواهد بود. در این موارد، گزینه‌های دیگری همچون بازسازی تابع یا حتی استفاده دقیق از goto cleanup; می‌تواند مورد استفاده قرار گیرد.

دستور بد نام goto

در زبان‌هایی غیر از C، نیاز کمی به goto به دلیل وجود راه‌های زیاد جایگزین برای انجام کار مد نظر، وجود دارد. همچنین این دستور به این دلیل که سریعاً از دست در می‌رود و دنبال کردن کد را سخت می‌کند، بدنام شده است.

با این حال هنوز هم می‌توان استفاده از دستور goto را در برخی از نسخه‌های پروژه‌های C مشاهده کرد که مهمترین آن‌ها هسته لینوکس است. قبل از اینکه همه کاربردهای goto را افتراض تلقی کنید، بهتر است تشریح کنیم که چرا برخی از استفاده‌های goto بهتر از دیگر روش‌ها است.

ساده‌ترین و بی ضررترین استفاده از goto، با یک exit تکی در پایین یک تابع نشان داده می‌شود:

```
if (p == NULL) goto exit;
...
exit:
    fclose(file1);
    fclose(file2);
...
return;
```

اگر در اینجا، تنها استفاده از goto مجاز باشد، این دستور مشکل زیادی نخواهد داشت.

احتمالاً مشکلات زمانی رخ می‌دهد که چندین هدف برای goto وجود داشته باشد، مخصوصاً اگر مسیر آن‌ها مختلف باشد. به طور خاص هرچند goto می‌تواند باعث پیشرفت در کدهای اسپاگتی شود ولی در عین حال، می‌تواند با حلقه‌های ساختاری جایگزین شود. اما به یاد داشته باشید که در بیشتر اوقات باید از goto اجتناب کنید.

^۱ trigger

تو در تو^۱ بودن را حداقل کنید

درک کردن کدهای توی در توی زیاد، دشوار است. هر سطح از مرحله تو در تو، شرط‌هایی اضافی را بر پشتۀ ذهنی^۲ خواننده تحمیل می‌کند. هنگامی که خواننده یک براکت پسته «{» را می‌بیند، انجام عملیات pop روی پشتۀ ذهنی شخص و اینکه شروط تحت آن را به یاد آورد، کار سختی خواهد بود.

در اینجا یک نمونه نسبتاً ساده از این موضوع وجود دارد. دقت کنید که برای فهمیدن محتواي داخلی کدام شرط، آن را دو مرتبه بررسی می‌کنید:

```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);
}
reply.Done();
```

زمانی که شما اولین براکت پسته «{» را می‌بینید، مجبور خواهید بود با خودتان فکر کنید، وااا! permission_result == SUCCESS تازه تمام شد، پس حالاً permission_result != SUCCESS است و این هنوز داخل بلوك کدی است که user_result == SUCCESS است.

همان گونه که مشاهده می‌کنید مجبور هستید که مقدار permission_result و user_result را همواره در ذهن خود نگه دارید و به محض اینکه هر بلوك { } پسته می‌شود، باید مقدارهای داخل ذهن خود را تغییر دهید.

این بخش خاص کد حتی بدتر است، زیرا شرایط بین SUCCESS و non-SUCCESS را به صورت متناسب نگه‌داری می‌کند.

^۱ Minimize Nesting

^۲ mental stack

چگونه تودرتو شدن‌ها روی هم انباسه^۱ می‌شود

قبل از تلاش برای اصلاح کد مثال قبلی بباید درباره چگونگی پایان یافتن آن صحبت کنیم. در ابتدا کد ساده بود:

```
if (user_result == SUCCESS) {
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);
}
reply.Done();
```

این کد کاملاً قابل فهم بوده و مشخص می‌کند که چه رشته خطایی باید نوشته شود و سپس با یک پاسخ تمام می‌شود.

اما بعد از این عملیات دوم به کد اضافه شده است:

```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
...
}
```

این تغییر با معنی است یعنی برنامه‌نویس یک تکه کد جدید برای درج کردن داشت و ساده‌ترین مکان را برای درج آن، پیدا کرد. این کد جدید، در ذهن او تازه و به صورت «برجسته^۲» بود و «تفاوت» این تغییر خیلی واضح است. البته این کار یک تغییر ساده به نظر می‌رسد.

اما هنگامی که بعدها شخص دیگری کد را بررسی می‌کند، این پیش زمینه را ندارد. او نیز همچون شما که برای اولین بار کد را در ابتدای این بخش خواندید، مجبور است که با یک نگاه همه کد را متوجه شود.

^۱ Accumulates

^۲ Bold

کلید طلایی

هنگام ایجاد تغییر، به کد خود به دید یک چشم انداز تازه نگاه کنید. برگردید و به آن به صورت کلی نگاه کنید.

حذف تودرتوها با برگرداندن زودتر یک نتیجه

حال بباید کد را بهبود دهیم. تودرتوهای شبیه این کد را می‌توان با مدیریت حالت‌های شکست^۲ در اولین فرصت ممکن حذف نموده و مقدار برگشتی تابع را سریع‌تر بازگرداند:

```
if (user_result != SUCCESS) {
    reply.WriteErrors(user_result);
    reply.Done();
    return;
}
if (permission_result != SUCCESS) {
    reply.WriteErrors(permission_result);
    reply.Done();
    return;
}
reply.WriteErrors("");
reply.Done();
```

به جای دو سطح، کد جدید تنها یک سطح از تودرتو بودن را دارد، و مهم‌تر از همه اینکه خواننده نیازی به عملیات pop در پشته مغز خود ندارد چرا که هر بلوك if با برگشت دادن یک مقدار به پایان می‌رسد.

حذف تودرتویی داخل حلقه‌ها

تکنیک «برگشت دادن یا return در اولین فرصت» همیشه قابل انجام نیست. به عنوان مثال در اینجا موردی از تودرتو بودن کد در حلقه را داریم:

```
for (int i = 0; i < results.size(); i++) {
    if (results[i] != NULL) {
        non_null_count++;
        if (results[i]->name != "") {
            cout << "Considering candidate..." << endl;
            ...
        }
    }
}
```

^۱ perspective

^۲ failure cases

}

در داخل حلقه، روش مشابه برای برگشت دادن زود هنگام مقدار، استفاده از دستور continue است:

```
for (int i = 0; i < results.size(); i++) {  
    if (results[i] == NULL) continue;  
    non_null_count++;  
    if (results[i]->name == "") continue;  
    cout << "Considering candidate..." << endl;  
    ...  
}
```

به همان روشی که یک return; if (...) به عنوان حفاظ جزئی¹ برای تابع عمل می‌نماید، دستور if (...) نیز به عنوان یک حفاظ جزئی برای حلقه عمل می‌کند.

هر چند به طور کلی دستور continue می‌تواند دستور goto در حلقه، گیج کننده باشد (زیرا این دستور ذهن خواننده را به اطراف منحرف می‌کند). اما در این مورد، به دلیل مستقل بودن هر تکرار از حلقه، خواننده می‌تواند به سادگی ببیند که continue فقط به معنی پرسش^۲ از این آیتم است.

' guard clause

skip

آیا می‌توانید جریان اجرایی برنامه را دنبال کنید؟

THREE-CARD MONTE



این فصل درباره کنترل جریان سطح-پایین^۱ بود یعنی نحوه ساختن حلقه‌ها، شرط‌ها و دیگر پرسش‌ها به گونه‌ای که خواندن آن‌ها، ساده باشد. اما این کافی نبوده و شما باید در مورد جریان برنامه خود در سطح بالا نیز فکر کنید. در حالت ایده‌آل باید بتوانید به راحتی کل مسیر اجرای برنامه خود را دنبال کنید یعنی از (main) شروع کرده و سپس از نظر ذهنی کد را دنبال نمایید و نیز به عنوان یک تابع، دیگر توابع را صدا بزنید تا اینکه در نهایت از برنامه خارج شوید.

با این حال، زبان‌های برنامه‌نویسی و کتابخانه‌ها در عمل دارای ساختارهای^۲ هستند که به کد اجازه اجرا شدن در پشت صحنه را می‌دهند و این دنبال کردن آن‌ها را سخت می‌کند. به این مثال‌ها توجه کنید:

ساختار برنامه‌نویسی	چگونگی ایجاد ابهام در جریان برنامه سطح بالا
threading	واضح نیست که کد چه زمانی اجرا می‌شود.
signal/interrupt handlers	کد خاصی ممکن است در هر زمانی اجرا شود.
exceptions	توقف برنامه می‌تواند با فراخوانی همزمان چندین تابع انجام شود.

^۱ low-level

^۲ constructs

به سختی می‌توان گفت که دقیقاً چه کدی در حال اجرا است! زیرا این کار در زمان کامپایل مشخص نیست.	function pointers & anonymous functions
ممکن است کد یک <code>object.virtualMethod()</code> زیرکلاس نامشخص را <code>invoke</code> کند.	Virtual methods

برخی از این ساختارها بسیار مفید بوده و حتی می‌توانند خوانایی کد شما را بیشتر نموده و افزونگی آن را کاهش دهند. اما به عنوان یک برنامه‌نویس، گاه با بی‌ملاحظه‌گی و بدون درک اینکه چقدر در آینده سبب دشواری فهمیدن کد برای خواننده‌ها می‌شوند، بیش از اندازه از آن‌ها استفاده می‌کنیم. همچنانی از سوی دیگر این ساختارها سبب سخت‌تر شدن رديابی اشکالات برنامه می‌شوند.

نکته اصلی این است که اجازه ندهید درصد زیادی از کد شما را این ساختارها تشکیل دهند. اگر از این ویژگی‌ها درست استفاده نکنید، ممکن است رديابی در سراسر کد شما را، به بازی Three-Card Monte تبدیل نمایند.

خلاصه فصل

کارهای زیادی وجود دارد که با انجام آن‌ها می‌توانید خواندن کنترل جریان کدهایتان را ساده‌تر کنید.

هنگام نوشتن یک مقایسه (while (`bytes_expected > bytes_received`)) بهتر است مقدار متغیر را در سمت چپ و مقداری که کمتر تغییر می‌کند را در سمت راست بنویسید (while (`bytes_received < bytes_expected`)).

همچنین می‌توانید ترتیب بلوک‌های دستور if/else را تغییر دهید. به طور کلی، تلاش کنید که موردهای مثبت یا جالب‌تر را برای نوشتن اولویت دهید. البته گاهی اوقات این معیارها با هم تداخل دارند، اما در صورت عدم تداخل، این یک قانون خوب است که باید از آن پیروی کنید.

برخی از ساختارهای برنامه‌نویسی مانند عملگر سه تایی (?:)، حلقه do/while و goto اغلب سبب می‌شوند که کد غیرقابل خواندن شود. بهتر است از آن‌ها را جایگزین کنید، زیرا همیشه گزینه‌های واضح‌تری وجود دارد.

برای دنبال کردن بلوک‌های کد تو در تو نیازمند تمرکز بیشتری هستید. هر تودرتوی جدید مستلزم ایجاد پیش زمینه بیشتر برای خواننده است تا بتواند کد را در پشته حافظه خود قرار دهد. برای اجتناب از تودرتو نوشتن عمیق، از کد خطی یا linear استفاده کنید.

به طور کلی برگرداندن سریع می‌تواند به حذف تودرتو بودن کد و پاک سازی آن کمک کند. در خصوص این مورد دستورات محافظتی^۱ خیلی مفید هستند (البته موارد ساده را در قسمت بالای تابع مدیریت کنید).

^۱ Guard statements

فصل هشتم

شکستن عبارت‌های غول پیکر به قسمت‌های کوچک



ماهی مرکب غول پیکر، حیوانی شگفت انگیز و باهوش است، اما خلقت تقریباً کامل آن، یک نقص کشنده دارد چرا که مغز دونات شکل^۱ این ماهی به دور مری^۲ آن پیچیده شده است و اگر به یکباره غذای زیادی ببلعد، باعث آسیب رسیدن به مغزش می‌شود.

ممکن است بپرسید این چه ربطی به کد دارد؟ پاسخ این است که کد نوشته شده در تکه‌های بسیار بزرگ می‌تواند اثر مشابه‌ای را داشته باشد. تحقیقات اخیر حاکی از آن است که اکثر ما در یک لحظه می‌توانیم، تنها به سه یا چهار چیز فکر کنیم^۳. این به بیان ساده‌تر، یعنی هرچه عبارت‌های کد بزرگ‌تر باشد، درک آن سخت‌تر خواهد بود.

کلید طلای

عبارت‌های غول پیکر خود را به قطعات قابل هضم‌تر تقسیم کنید.

در این فصل روش‌های مختلفی را برای دستکاری و تجزیه کد به شما خواهیم آموخت تا بلعیدن آن‌ها ساده‌تر شود.

متغیرها را شرح دهید

ساده‌ترین روش برای تجزیه یک عبارت، معرفی متغیرهای اضافه‌ای است که یک زیرعبارت^۴ کوچک‌تر را تولید می‌کند. این متغیر اضافی گاه «متغیر توضیح دهنده^۵» نامیده می‌شود، زیرا کمکی در جهت توضیح معنای زیرعبارت است. به عنوان مثال کد زیر را در نظر بگیرید:

```
if line.split(':')[0].strip() == "root":  
    ...
```

و حال این کد مشابه را که همراه با متغیرهای توضیح دهنده است را مشاهده نمایید:

```
username = line.split(':')[0].strip()  
if username == "root":  
    ...
```

^۱ donut-shaped

^۲ esophagus

^۳ Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24, 97–185.

^۴ subexpression

^۵ explaining variable

متغیرهای خلاصه^۱

حتی اگر یک عبارت نیازمند توضیح نباشد(البته به این دلیل که می‌توانید معنای آن را بفهمید)، باز هم گنجاندن آن عبارت در یک متغیر جدید مفید خواهد بود. البته اگر هدف از این کار صرفاً جایگزینی آن کد با نامی کوچکتر است تا بتوان آن را راحت‌تر مدیریت یا به آن فکر کرد، این را یک متغیر خلاصه می‌نامیم.

به عنوان مثال عبارت زیر را در این کد در نظر بگیرید:

```
if (request.user.id == document.owner_id) {
    // user can edit this document...
}

...
if (request.user.id != document.owner_id) {
    // document is read-only...
```

عبارة request.user.id == document.owner_id شاید بزرگ به نظر نرسد، اما دارای پنج متغیر است که باعث می‌شود زمان مورد نیاز برای فکر کردن به آن بیشتری شود.

جمله «آیا کاربر مالک سند است؟»، مفهوم اصلی این کد است که با اضافه کردن یک متغیر خلاصه، این مفهوم می‌تواند به صورت روشن‌تری بیان شود:

```
final boolean user_owns_document = (request.user.id == document.owner_id);
if (user_owns_document) {
    // user can edit this document...
}
...
if (!user_owns_document) {
    // document is read-only...
}
```

ممکن است این تغییر، چیز زیادی به نظر نرسد، اما دستور if (user_owns_document) برای فکر کردن، کمی ساده‌تر است. همچنین تعریف user_owns_document در ابتدا کد به خواننده می‌گوید که «این مفهومی است که ما در طی این تابع به آن خواهیم پرداخت.».

^۱ Summary Variables

استفاده از قوانین دمورگان^۱

اگر درس مدار یا منطق را گذرانده باشید احتمالاً قوانین دمورگان را به خاطر دارید. این قوانین دو راه برای نوشتتن یک عبارت Boolean به شکل معادل آن هستند:

- 1) $\text{not } (\text{a or b or c}) \Leftrightarrow (\text{not a}) \text{ and } (\text{not b}) \text{ and } (\text{not c})$
- 2) $\text{not } (\text{a and b and c}) \Leftrightarrow (\text{not a}) \text{ or } (\text{not b}) \text{ or } (\text{not c})$

گاه می‌توانید با استفاده از این قوانین خوانایی بیشتری در یک متغیر Boolean ایجاد کنید. به عنوان نمونه اگر کد شما به صورت زیر باشد:

```
if (!(file_exists && !is_protected)) Error("Sorry, could not read file.");
```

می‌توانید آن را به شکل زیر بنویسید:

```
if (!file_exists || is_protected) Error("Sorry, could not read file.");
```

سوءاستفاده از منطق اتصال کوتاه^۲

در اکثر زبان‌های برنامه‌نویسی، عملگرهای Boolean ارزیابی اتصال کوتاه را انجام می‌دهند. به عنوان مثال، دستور $(\text{a} \parallel \text{b})$ در صورتی که مقدار a برابر true باشد، مقدار b را بررسی نمی‌کند. هر چند این رفتار بسیار مفید است اما گاهی می‌تواند برای تحقق منطق پیچیده به شکل صحیح مورد استفاده قرار نگیرد.

در اینجا مثالی از یک دستور تکی داریم:

```
assert((!bucket = FindBucket(key))) || !bucket->IsOccupied());
```

در زبان انگلیسی، آنچه که این کد می‌گوید این است: «برای این key مقدار bucket را بگیر، اگر مقدار bucket برابر null نبود، پس مطمئن شوید که اشغال نشده است». هر چند این فقط یک خط کد است ولی واقعاً سبب توقف اکثر برنامه‌نویسان می‌شود تا کمی در مورد آن فکر کنند. اکنون آن را با کد زیر مقایسه کنید:

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

^۱ De Morgan

^۲ Short-Circuit

این یکی حتی اگر دو خط کد باشد دقیقاً همان کار را انجام داده و درک آن بسیار ساده‌تر است.

ممکن است سوال کنید پس چرا در نسخه اول، این کد به صورت یک عبارت غول پیکر نوشته شده است؟ ممکن است بگوییم شاید در آن زمان، برنامه‌نویس احساس باهوشی می‌کرده و برایش لذت خاصی در تقسیم کردن منطق به یک قطعه کد مختصراً وجود داشته است. البته این قابل درک است و مانند حل کردن یک پازل نقاشی است که همه ما دوست داریم در محل کار تفریح هم داشته باشیم ولی مشکل این است که این نوع کد برای کسانی که می‌خواهند آن را بخوانند، یک دست انداز ذهنی در برابر سرعت خواندن کد ایجاد می‌کند.

کلید طلایی

مراقب قطعه کدهای هوشمندانه^۱ باشید. آن‌ها معمولاً کسانی را که در آینده آن را می‌خوانند دچار سردگمی می‌کنند.

حال ببینیم که آیا این بدان معنی است که شما باید از کاربرد رفتار اتصال کوتاه^۲ خودداری کنید؟ خیر، موارد زیادی وجود دارد که می‌توان از آن به شکل شفاف استفاده کرد، همچون:

```
if (object && object->method()) ...
```

شیوه جدیدتری نیز در اینجا وجود دارد که قابل ذکر است: در زبان‌هایی مثل JavaScript، Python و Ruby عملگر or یکی از آرگومان‌ها را بر می‌گرداند (یعنی آن را به Boolean تبدیل نمی‌کند). بنابراین کد زیر:

```
x = a || b || c
```

می‌تواند برای انتخاب اولین مقدار صحیح از a، b یا c استفاده شود.

مثال: کشمکش^۳ با منطق پیچیده

فرض کنید که شما در حال پیاده سازی کلاس Range به این صورت هستید:

```
struct Range {
    int begin;
    int end;
    // For example, [0,5) overlaps with [3,8)
    bool OverlapsWith(Range other);
```

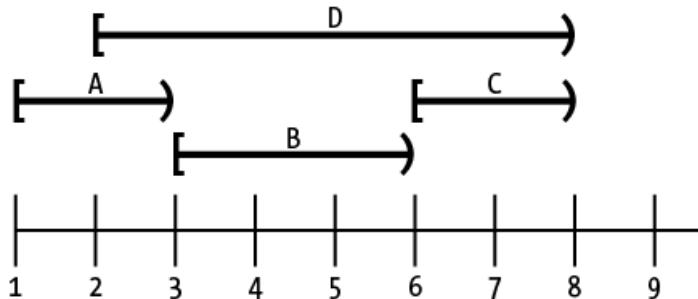
^۱ clever

^۲ short-circuit

^۳ Wrestling

```
};
```

تصویر زیر چند نمونه از محدوده‌ها^۱ را نشان می‌دهد:



توجه کنید که end یک کلمه غیرفراغیر^۲ است بنابراین A، B و C با یکدیگر همپوشانی^۳ نداشته اما D با همه همپوشانی دارد.

یک تلاش برای پیاده سازی این وضعیت به این صورت است که قرار گیری هر نقطه انتهایی از یک محدوده در محدوده دیگر را بررسی می‌کند:

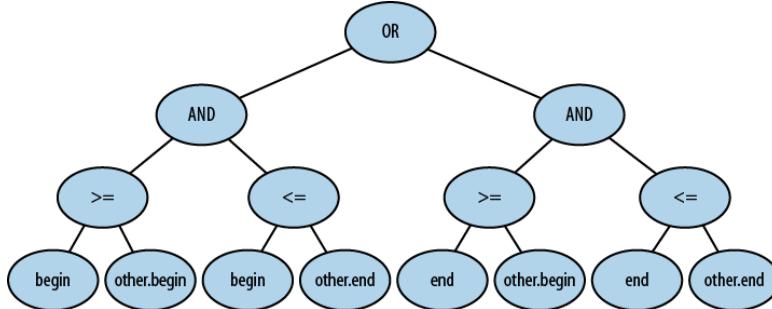
```
bool Range::OverlapsWith(Range other) {
    // Check if 'begin' or 'end' falls inside 'other'.
    return (begin >= other.begin && begin <= other.end) ||
           (end >= other.begin && end <= other.end);
}
```

ولو اینکه این کد تنها دو خط طولانی باشد، باز هم موارد بسیار زیادی در آن بررسی می‌شود. در تصویر زیر همه منطق موجود در این کد نشان داده شده است.

^۱ ranges

^۲ noninclusive

^۳ overlaps



موارد و شرایط بسیاری وجود دارد که باید درباره آن فکر شود، چرا که رخدادن سه‌های یک اشکال در این مورد راحت است. صحبت از این است که یک باگ وجود دارد. زیرا کد قبلی ادعای همپوشانی محدوده [0,2] را دارد در حالی که در واقعیت این چنین نیست.

مشکل این است که باید در زمان مقایسه مقدارهای begin/end با استفاده از `<=>` یا `<>` مراقب باشید. نمونه اصلاح شده آن را به این شکل است:

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end);
```

اگرچه این مشکل برطرف شد ولی یک اشکال دیگر نیز وجود دارد! این کد مواردی که begin/end به صورت کامل دیگر موارد را پوشش می‌دهند، نادیده می‌گیرد. در اینجا نمونه اصلاح شده که این مورد را نیز مدیریت می‌کند، مشاهده می‌کنید:

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end) ||
       (begin <= other.begin && end >= other.end);
```

این کد بسیار پیچیده شده است. شما نمی‌توانید انتظار داشته باشید کسی بعد از خواندن این کد با اطمینان بگوید که صحیح است. پس باید چه کاری انجام دهیم؟ چگونه می‌توانیم این عبارت غول‌پیکر را تجزیه کنیم؟

پیدا کردن یک رویکرد ظرفیتر

این نمونه یکی از موقوعی است که باید توقف کرده و رویکرد متفاوتی را در نظر بگیرید. آنچه به عنوان یک مشکل ساده شروع شد (یعنی بررسی اینکه دو محدوده با یکدیگر همپوشانی دارند یا نه)

در نهایت به یک منطق پیچیده تعجب آور تبدیل گشت. این اغلب نشانه‌ای است برای این مطلب که: باید راهی آسان‌تر وجود داشته باشد.

اما پیدا کردن یک راه حل ظریف‌تر نیازمند خلاقیت است. یک روش این است که ببینید آیا می‌توانید مسئله را در جهت عکس^۱ حل کنید یا نه؟ بسته به موقعیتی که در آن قرار دارید، این کار می‌تواند با تکرار از طریق آرایه‌ها، به صورت معکوس و یا پر کردن برخی از ساختارداده^۲ از انتهای^۳ (به جای از ابتداء^۴) انجام شود.

در اینجا معکوس OverlapsWith() به معنی «همپوشانی ندارد» است. تشخیص این موضوع که دو محدوده با هم همپوشانی نداشته باشند، مسئله‌ای ساده‌تر است، زیرا تنها دو امکان وجود دارد:

۱. محدوده دیگر قبل از شروع این محدوده به پایان می‌رسد.

۲. محدوده دیگر پس از پایان این محدوده شروع می‌شود.

ما می‌توانیم این مطلب را به سادگی در کد پیاده‌سازی کنیم:

```
bool Range::OverlapsWith(Range other) {
    if (other.end <= begin) return false; // They end before we begin
    if (other.begin >= end) return false; // They begin after we end
    return true; // Only possibility left: they overlap
}
```

هر خط از کد بسیار ساده بوده و تنها شامل یک مقایسه تکی است. این باعث می‌شود تا خواننده توان ذهنی خود را صرف این مسئله کند که آیا \Rightarrow صحیح است یا خیر؟

^۱ opposite

^۲ data structure

^۳ backward

^۴ forward

شکستن دستورات غول‌پیکر

هرچند این فصل درباره شکستن عبارت‌های تکی^۱ است، اما همین روش‌ها، برای شکستن دستورات بزرگتر نیز کاربرد دارد. به عنوان مثال، موارد موجود در کد JavaScript زیر بیش از آن است که در یک نگاه بتوان همه آن‌ها را درک کرد:

```
var update_highlight = function (message_num) {
    if ($("#vote_value" + message_num).html() === "Up") {
        $("#thumbs_up" + message_num).addClass("highlighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    } else if ($("#vote_value" + message_num).html() === "Down") {
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).addClass("highlighted");
    } else {
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    }
};
```

با اینکه عبارت‌های تکی در این کد چندان بزرگ نیستند، ولی از کنار هم قرار گرفتن آن‌ها، دستور غول‌پیکری شکل می‌گیرد که ممکن است سبب تعجب شما در یک لحظه شود. البته جای نگرانی نیست چرا که خوشبختانه به دلیل تشابه بسیاری از عبارت‌ها می‌توانیم آن‌ها را به عنوان یک متغیر خلاصه، در بالای تابع قرار دهیم(همچنین این کار نمونه‌ای از اصل DRY (یا خودت را تکرار نکن) است):

```
var update_highlight = function (message_num) {
    var thumbs_up = $("#thumbs_up" + message_num);
    var thumbs_down = $("#thumbs_down" + message_num);
    var vote_value = $("#vote_value" + message_num).html();
    var hi = "highlighted";
    if (vote_value === "Up") {
        thumbs_up.addClass(hi);
        thumbs_down.removeClass(hi);
    } else if (vote_value === "Down") {
        thumbs_up.removeClass(hi);
```

^۱ individual

```

        thumbs_down.addClass(hi);
    } else {
        thumbs_up.removeClass(hi);
        thumbs_down.removeClass(hi);
    }
};

```

با اینکه لزومی به ساختن "var hi = "highlighted" نیست، اما از آنجا که شش نسخه از آن وجود دارد، مزایای قانع کننده‌ای دارد، از جمله اینکه:

- کمک می‌کند تا از اشتباهات تایپی جلوگیری شود (آیا در مثال اول متوجه اشتباه تایپی رشته highlighted در مورد پنجم شدید؟)
- سبب کوتاهتر شدن عرض خط و در نتیجه ساده‌تر شدن بررسی کد در یک نگاه می‌شود.
- اگر نام کلاس نیاز به تغییر داشته باشد، فقط یک مکان برای تغییر آن وجود دارد.

یکی دیگر از روش‌های خلاقانه برای ساده کردن یک عبارت

در زیر مثال دیگری در زبان C++ که در هر عبارت آن اتفاقات زیادی می‌افتد را مشاهده می‌کنید:

```

void AddStats(const Stats& add_from, Stats* add_to) {
    add_to->set_total_memory(add_from.total_memory() + add_to->total_memory());
    add_to->set_free_memory(add_from.free_memory() + add_to->free_memory());
    add_to->set_swap_memory(add_from.swap_memory() + add_to->swap_memory());
    add_to->set_status_string(add_from.status_string() + add_to->status_string());
    add_to->set_num_processes(add_from.num_processes() + add_to->num_processes());
    ...
}

```

بار دیگر، چشمان شما با کدی دارای عبارات طولانی و مشابه هم (اما نه دقیقاً یکسان) رو برو شده است. پس از ده ثانیه بررسی دقیق، ممکن است متوجه شوید که هر خط یک کار مشابه را در فیلدهای متفاوت انجام می‌دهد:

```
add_to->set_XXX(add_from.XXX() + add_to->XXX());
```

در C++ می‌توانیم برای پیاده سازی چنین چیزی، از ماکروها^۱ استفاده کنیم:

```
void AddStats(const Stats& add_from, Stats* add_to) {
    #define ADD_FIELD(field) add_to->set_##field(add_from.field() + add_to-
>field())
    ADD_FIELD(total_memory);
    ADD_FIELD(free_memory);
    ADD_FIELD(swap_memory);
    ADD_FIELD(status_string);
    ADD_FIELD(num_processes);
    ...
    #undef ADD_FIELD
}
```

اکنون که همه قسمت‌های به هم پیچیده را از بین بردیم، می‌توانیم بلا فاصله با یک نگاه ماهیت اینکه چه اتفاقی می‌افتد را درک کنیم. کاملاً واضح است که هر خط از کد کار مشابه‌ای را انجام می‌دهد.

البته توجه داشته باشید که ما همیشه از کاربرد ماکروها دفاع نمی‌کنیم. در واقع اصل اولیه اجتناب از آن‌ها است زیرا ماکروها می‌توانند سبب ایجاد سردرگمی در کد شده و معرف باگ‌های ظریفی باشند. اما گاهی اوقات همچون این مورد، به دلیل ساده بودن می‌توانند مزیت واضحی را برای خوانایی بهتر کد فراهم کنند.

خلاصه فصل

فکر کردن در مورد عبارات غول پیکر سخت است. این فصل روش‌هایی را برای شکستن آن‌ها به قسمت‌های کوچک‌تر نشان داد، که خواننده بتواند آن‌ها را تکه به تکه هضم کند.

یک روش ساده معرفی متغیرهای خلاصه است که مقدار بعضی از زیرعبارات بزرگ را در خود نگه می‌دارند. این کار سه مزیت دارد:

- عبارت غول پیکر را به تکه‌هایی کوچک‌تر تقسیم می‌کند.
- کد را با توصیف زیرعبارات در یک نام مختصر، مستند می‌کند.

¹ macro

• به خواننده کمک می‌کند تا مفاهیم اصلی کد را شناسایی کند.

راه حل دیگر دستکاری منطق خود با استفاده از قوانین دمورگان است. این روش گاهی می‌تواند یک عبارت Boolean را به شیوه‌ای واضح‌تر بازنویسی کند. (مثلاً تغییر عبارت `((a && !b) if (!a || b)`).

ما مثالی که در آن منطق شرطی پیچیده، در دستورات کوچکی شبیه "if (a < b)... if (a < b)" شکسته شد را نشان دادیم. در حقیقت همه کدهای بهبود داده شده در این فصل، دارای دستورات if بودند که بیشتر از دو مقدار داخل آن‌ها نبود. این‌ها مثال‌هایی برای تمرین بودند و ممکن است همیشه انجام چنین کاری امکان پذیر نباشد. گاهی این کار به بی‌اثر^۱ کردن مشکل یا در نظر گرفتن حالت معکوس هدف شما، نیاز دارد.

سخن پایانی اینکه اگرچه این فصل درباره روش شکستن عبارتهای تکی بود، ولی این روش‌ها اغلب امکان استفاده برای بلوک‌های بزرگ را نیز دارند. بنابراین هر کجا با آن‌ها روبرو شدید برای شکستن منطق پیچیده به قسمت‌های کوچک‌تر، دریغ نکنید.

^۱ negating

فصل نهم

متغیرها و خوانایی



در این فصل خواهید دید که چگونه استفاده شلخته^۱ از متغیرها سبب سخت‌تر شدن درک برنامه می‌شود. در اینجا به طور خاص با سه مشکل اساسی رویرو هستیم:

۱. هر چه متغیرهای بیشتری وجود داشته باشد، دنبال کردن همه آن‌ها سخت‌تر می‌شود.
 ۲. هرچه محدوده یک متغیر بزرگ‌تر باشد، شما باید پیگیری طولانی‌تری را در مورد آن انجام دهید.
 ۳. هرچه یک متغیر بیشتر تغییر کند، پیگیری مقدار فعلی آن سخت‌تر می‌شود.
- در ادامه و طی سه بخش نحوه مقابله با این مسائل سه گانه بررسی خواهد شد.

از بین بردن^۲ متغیرها

همان گونه که به یاد دارید در فصل هشتم نشان دادیم که معرفی متغیر توضیح‌دهنده^۳ یا خلاصه^۴ می‌تواند سبب خوانایی بیشتر کد شود. این متغیرها مفید بودند، زیرا عبارت‌های غول پیکر را تجزیه نموده و به عنوان شکلی از مستندات عمل می‌کردند.

اما در این بخش می‌خواهیم متغیرهایی که باعث بهبود خوانایی کد نمی‌شوند را از بین بیریم. با حذف متغیر غیرضروری، کد جدید مختصرتر شده و به آسانی قابل درک خواهد شد.

متغیرهای موقتی^۵ بی فایده

در کد Python زیر، متغیر now را در نظر بگیرید:

```
now = datetime.datetime.now()
root_message.last_view_time = now
```

به نظر می‌رسد به سه دلیل، نگه داشتن متغیر now ارزش نداشته باشد:

- این متغیر عبارت پیچیده‌ای را تجزیه نمی‌کند.
- شفافیت خاصی را اضافه نمی‌کند، چرا که عبارت datetime.datetime.now() به اندازه کافی واضح است.

^۱ sloppy

^۲ Eliminating Variables

^۳ explaining

^۴ summary

^۵ Temporary

- از آنجا که این متغیر فقط یکبار استفاده شده است، بنابراین هیچ کد افزونگی را فشرده نمی‌کند.

بدون متغیر now کد به راحتی قابل درک است:

```
root_message.last_view_time = datetime.datetime.now()
```

متغیرهایی مانند now معمولاً «ماند» هستند که بعد از ویرایش کد همچنان باقی می‌مانند. این متغیر ممکن است در ابتدا در چندین مکان استفاده شده باشد و یا حتی ممکن است کدنویس پیش بینی می‌کرده که از now چندین بار استفاده خواهد کرد، اما در ادامه هرگز به آن نیازی پیدا نکرده است.

از بین بردن نتایج واسطه^۲



در اینجا مثالی از کد یک تابع JavaScript داریم که مقداری را از یک آرایه حذف می‌کند:

```
var remove_one = function (array, value_to_remove) {
    var index_to_remove = null;
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            index_to_remove = i;
            break;
        }
    }
    if (index_to_remove !== null) {
        array.splice(index_to_remove, 1);
    }
}
```

¹ leftovers

² Intermediate

};

متغیر index_to_remove تنها برای نگهداشتن نتیجه واسطه (میانی) استفاده شده است. گاهی چنین متغیرهایی می‌توانند به محض دریافت نتیجه، حذف شوند:

```
var remove_one = function (array, value_to_remove) {
  for (var i = 0; i < array.length; i += 1) {
    if (array[i] === value_to_remove) {
      array.splice(i, 1);
      return;
    }
  }
};
```

با اجازه به کد برای برگرداندن سریع، می‌توانیم از دست index_to_remove خلاص شده و کد را کمی ساده کنیم. به طور کلی این موضوع استراتژی خوبی است که «یک کار را در اولین فرصت به اتمام برسانید».

از بین بردن متغیرهای کنترل جریان

گاهی اوقات، الگوی زیر را در حلقه‌های کد خود می‌بینید:

```
boolean done = false;
while /* condition */ && !done) {
  ...
  if (...) {
    done = true;
    continue;
  }
}
```

متغیر done ممکن است از طریق حلقه در چندین مکان، برابر true مقداردهی شود. در اکثر اوقات کدی مانند این برای برآورده کردن برخی قوانین نانوشته است که «باید در وسط یک حلقه، از برنامه جدا شد». و این در حالی است که اصولاً چنین قانونی وجود ندارد!

متغیرهایی شبیه done همان چیزی است که ما آن را متغیرهای کنترل جریان می‌نامیم. تنها هدف آن‌ها هدایت اجرای برنامه است و شامل هیچ‌گونه داده واقعی از برنامه نمی‌باشند. تجربه به

ما ثابت کرده که متغیرهای کنترل جریان اغلب می‌توانند با استفاده بهتر از برنامه‌نویسی ساخت‌یافته^۱ حذف شوند:

```
while /* condition */ {
    ...
    if (...) {
        break;
    }
}
```

در این مثال حذف این مشکل بسیار آسان است، اما اگر یک تجزیه ساده برای چندین حلقه تودرتو کافی نباشد باید چه کرد؟ در اینگونه موارد راه حل جابجایی کد به یک تابع جدید است (جابجایی کد داخل حلقه یا کل حلقه).

آیا می‌خواهید همکاران تان احساس کنند که همیشه در وضعیت «زمان مصاحبه» قرار دارند؟

Eric Brechner در شرکت مایکروسافت درباره این مورد که چگونه یک سوال عالی در زمان مصاحبه باید شامل حداقل سه متغیر باشد^۲ صحبت کرده است. احتمالاً به این دلیل که برخورد با سه متغیر به صورت هم زمان مصاحبه شونده را وادار می‌کند که سخت فکر کند! این برای مصاحبه منطقی است، جایی که شما تلاش می‌کنید یک گزینه را محدود کنید. اما آیا می‌خواهید همکاران تان هنگام خواندن کد شما احساس کنند که در جلسه مصاحبه حضور دارند؟

دامنه^۳ متغیرهای خود را کوچک^۴ کنید

همه ما این توصیه را شنیده‌ایم که: «از متغیرهای سراسری^۵ اجتناب کنید». این توصیه خوبی است، زیرا ردیابی اینکه همه متغیرهای سراسری در کجا و چگونه استفاده شده‌اند، سخت است. در این مورد با «آلوده کردن فضای نام»^۶ یعنی قرار دادن یک دسته از نام‌ها در آن مکان، مشکل را حل نمود چرا که ممکن است با متغیرهای محلی شما تداخل پیدا کند و ممکن است کد هنگامی که قصد استفاده از یک متغیر محلی را دارد به صورت تصادفی یک متغیر سراسری را تغییر دهد (و یا برعکس).

^۱ structured programming

^۲ Eric Brechner's I. M. Wright's "Hard Code" (Microsoft Press, 2007), p. 166.

^۳ scope

^۴ Shrink

^۵ global

^۶ polluting the namespace

در حقیقت این ایده خوبی است که محدوده همه متغیرها (و نه فقط آنها که سراسری هستند) را کوچک کنید.

کلید طلایبی

متغیر خود را تا حد امکان با چند خط کد، قابل مشاهده کنید.

بسیاری از زبان‌های برنامه‌نویسی چندین سطح دامنه/دسترسی، افزودن ماژول، کلاس، تابع و محدوده بلوك را ارائه می‌دهند. به صورت یک قانون کلی، استفاده از دسترسی محدودتر بهتر است، زیرا بدان معنی است که متغیر را می‌توان با تعداد کمتری از خطوط کد مشاهده کرد.

دلیل انجام این کار این است که به شکل موثری، تعداد متغیرهای را که خواننده باید در یک زمان واحد به آن‌ها فکر کند، کاهش می‌دهد. اگر محدوده همه متغیرها را نصف کنید، میانگین تعداد متغیرها در محدوده در هر زمان نیز نصف خواهد شد.

به عنوان مثال فرض کنید یک کلاس خیلی بزرگ، با یک متغیر عضو دارید که فقط توسط دو متده به صورت زیر مورد استفاده قرار می‌گیرد:

```
class LargeClass {
    string str_;
    void Method1() {
        str_ = ...;
        Method2();
    }
    void Method2() {
        // Uses str_
    }
    // Lots of other methods that don't use str_ ...
};
```

به عبارت دیگر متغیر عضو کلاس مانند یک متغیر سراسری کوچک^۱ در داخل قلمرو^۲ کلاس است. در کلاس‌های بزرگ، پیگیری همه متغیرهای عضو و اینکه چه متده، کدام یک از آن‌ها را تغییر می‌دهد سخت است. هرچه تعداد این متغیرهای سراسری کوچک کمتر باشد، بهتر است. در این مثال منطقی است که متغیر str_ را به یک متغیر محلی تنزل دهیم:

^۱ mini-global

^۲ realm

```

class LargeClass {
    void Method1() {
        string str = ...;
        Method2(str);
    }
    void Method2(string str) {
        // Uses str
    }
    // Now other methods can't see str.
};

```

روش دیگر برای محدود کردن دسترسی اعضای کلاس این است که تا جای ممکن متدهای استاتیک^۱ ایجاد کنیم. متدهای استاتیک روشنی عالی برای اطلاع به خواننده است که «این خطوط کد، از دیگر متغیرها جداسازی^۲ شده‌اند».

همچنین رویکرد دیگر این است که کلاس بزرگ را به چندین کلاس کوچک‌تر تجزیه کنیم. البته این روش در صورتی مفید است که کلاس‌های کوچک‌تر را از یکدیگر جداسازی کنیم. اگر قرار بر ساخت دو کلاس باشد که به اعضای یکدیگر دسترسی داشته باشند، در واقع هیچ کاری انجام نداده‌اید.

این قانون در مورد تجزیه فایل‌های بزرگ به توابع بزرگ یا توابع کوچک‌تر نیز حاکم بوده و انگیزه مهمی برای انجام این کار جداسازی داده‌ها^۳ از جمله متغیرها است.

البته توجه داشته باشید که زبان‌های مختلف قوانین متفاوتی در مورد یک محدوده دقیقاً تا چه محدوده‌ای است دارند. در اینجا فقط به برخی از قوانین جالب توجه درباره محدوده متغیرها اشاره می‌کنیم.

محدوده دستور IF در زبان C++

فرض کنید شما کد C++ زیر را دارید:

```

PaymentInfo* info = database.ReadPaymentInfo();
if (info) {
    cout << "User paid: " << info->amount() << endl;
}

```

^۱ Static methods

^۲ isolated

^۳ data

```

}
// Many more Lines of code below ...

```

متغیر info برای ادامه تابع در محدوده باقی خواهد ماند، بنابراین خواننده این کد ممکن است را در ذهن خود نگه داشته و همواره نگران چگونگی استفاده مجدد این متغیر در ادامه باشد.

اما در C++ میتوانیم info را در داخل عبارت شرطی نیز تعریف کنیم، مانند کد زیر که info فقط داخل دستور if استفاده شده است.:

```

if (PaymentInfo* info = database.ReadPaymentInfo()) {
    cout << "User paid: " << info->amount() << endl;
}

```

به همین دلیل خواننده به راحتی میتواند info را خارج از محدوده آن فراموش کند.

ایجاد متغیرهای Private در JavaScript

متغیر سراسری که فقط در یک تابع استفاده شده است را در نظر بگیرید:

```

submitted = false; // Note: global variable
var submit_form = function (form_name) {
    if (submitted) {
        return; // don't double-submit the form
    }
    ...
    submitted = true;
};

```

یک متغیر سراسری مانند submitted میتواند سبب احساس وحشت در شخصی که کد را میخواند بشود. هرچند ظاهرا به نظر میرسد submit_form() تنها تابعی است که از submitted استفاده میکند اما شما نمیتوانید با اطمینان این را بگویید. در حقیقت، یک فایل JavaScript دیگر ممکن است از متغیر سراسری که submitted نامیده شده است، برای هدفی دیگر استفاده کند! شما میتوانید با قرار دادن submitted در داخل closure(بستار) از این مشکل جلوگیری کنید:

```

var submit_form = (function () {
    var submitted = false; // Note: can only be accessed by the function below
    return function (form_name) {
        if (submitted) {
            return; // don't double-submit the form
        }
    };
});

```

```

    }
    ...
    submitted = true;
};

}();

```

به پرانتزهای خط آخر توجه داشته باشید، تابع anonymous بلافاصله اجرا شده وتابع داخلی را بر می‌گرداند.

اگر تا کنون این روش را ندیده‌اید، ممکن است ابتدا عجیب به نظر برسد. این کار سبب می‌شود تا یک محدوده private که تنها تابع داخلی به آن دسترسی دارد، ایجاد شود. حال خواننده درباره موارد دیگر استفاده از submitted نگرانی ندارد. همچنین بابت ایجاد تداخل با دیگر متغیرهای سراسری با همین نام نیز نگران نخواهد بود.(برای مطالعه بیشتر درباره این روش می‌توانید به کتاب [JavaScript: The Good Parts by Douglas Crockford [O'Reilly, 2008] مراجعه نمایید).

دامنه سراسری JavaScript

در JavaScript اگر کلمه کلیدی var را در تعریف یک متغیر ننویسید(مثلاً به جای var x=1 بنویسید `<script>`، متغیر در دامنه سراسری تعریف می‌شود، جایی که هر فایل JavaScript و بلوک </> می‌تواند به آن دسترسی داشته باشد. به عنوان مثال:

```

<script>
var f = function () {
    // DANGER: 'i' is not declared with 'var'!
    for (i = 0; i < 10; i += 1) ...
};

f();
</script>

```

این کد اشتباهات را در محدوده جهانی قرار می‌دهد، بنابراین خارج از بلوک نیز امکان مشاهده آن وجود دارد:

```

<script>
    alert(i); // Alerts '10'. 'i' is a global variable!
</script>

```

بسیاری از برنامه‌نویسان از این قانون محدوده اطلاع نداشته و این می‌تواند باعث باگ‌های عجیبی شود. از جمله موقع آشکار شدن چنین باگی، هنگامی است که دوتابع، یک متغیر محلی را بنام مشابه ایجاد کرده، ولی استفاده از var را فراموش می‌کنند. این تابع‌ها ندانسته دچار تداخل (cross-talk) می‌شوند و برنامه‌نویس بیچاره احتمالاً نتیجه می‌گیرد که کامپیوتersh دچار اشکال شده یا RAM او خراب شده است!

بهترین تجربه^۱ برای JavaScript این است که **همیشه متغیرها را با استفاده از کلمه کلیدی var تعریف کنید**(مثلا 1 = var x). این کار دامنه متغیر را به تابعی که در آن تعریف شده است (یعنی داخلی‌ترین تابع) محدود می‌کند.

نحو محدوده‌ها در زبان JavaScript و Python

زبان‌هایی مانند C++ و Java دارای محدوده بلوک^۲ هستند، یعنی جایی که متغیرها داخل یک if، for، try یا ساختارهایی مشابه (که دامنه تودرتوی یک بلوک محدود شده است)، تعریف می‌شوند:

```
if (...) {
    int x = 1;
}
x++; // Compile-error! 'x' is undefined.
```

از سوی دیگر در زبان Python یا JavaScript متغیرهای تعریف شده در یک بلوک، در کل تابع قابل دسترس هستند. به عنوان مثال به استفاده از example_value در این کد پایتون که به شکل صحیح کار می‌کند توجه کنید:

```
# No use of example_value up to this point.
if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break
for logger in debug.loggers:
    logger.log("Example:", example_value)
```

^۱ best practice

^۲ block scope

این قانون محدوده، بسیاری از برنامهنویسان را متعجب کرده و فهمیدن کدی شبیه این را سخت می‌کند. در زبان‌های دیگر پیدا کردن جایی که برای اولین بار `example_value` تعریف شده است، ساده‌تر است. چرا که شما در گوشه سمت چپ، داخل تابع خود به دنبال تعریف آن می‌گردید.

این مثال اشکال دیگری نیز دارد چرا که اگر `example_value` در بخش اول کد تنظیم نشده باشد، بخش دوم پیغام خطای^۱ را به صورت `NameError: 'example_value' is not defined` ایجاد خواهد کرد. ما می‌توانیم این مشکل را برطرف کنیم و با تعریف `example_value` قبل از نقطه اشتراک آن‌ها(یعنی در عبارت‌های تودرتو)، کد را خواناتر کنیم:

```
example_value = None
if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break
if example_value:
    for logger in debug.loggers:
        logger.log("Example:", example_value)
```

با این حال، این موردی است که می‌توان `example_value` را حذف نمود چرا که فقط یک نتیجه میانی را نگه می‌دارد و همان گونه که در قسمت حذف نتایج میانی دیدیم، متغیرهایی شبیه به این، می‌توانند توسط «یک وظیفه را در اسرع وقت کامل کنید» حذف شوند. البته در این مورد به این معنی است که به محض پیدا کردن `example_value` مقدار آن را ثبت^۲ کنید. کد جدید شبیه عبارت زیر خواهد بود:

```
def LogExample(value):
    for logger in debug.loggers:
        logger.log("Example:", value)
if request:
    for value in request.values:
        if value > 0:
            LogExample(value) # deal with 'value' immediately
            break
```

^۱ exception

^۲ logging

زبان برنامه‌نویسی اصلی C الزام می‌کرد که تعریف همه متغیرها در بالای تابع یا بلوک کد باشند و این موضوع، تاسف بار بود، زیرا برای توابع طولانی با متغیرهای زیاد، خواننده را مجبور می‌کرد در یک لحظه درباره همه آن متغیرها فکر کند، حتی اگر تا مدت زمان زیادی مورد استفاده قرار نگرفته باشند(البته C99 و C++ این الزام را حذف کردند).

در مثال زیر تمام متغیرها در بالای تابع تعریف شده‌اند:

```
def ViewFilteredReplies(original_id):
    filtered_replies = []
    root_message = Messages.objects.get(original_id)
    all_replies = Messages.objects.select(root_id=original_id)
    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
    root_message.save()
    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
            filtered_replies.append(reply)
    return filtered_replies
```

این کد خواننده را مجبور می‌کند تا هم‌زمان به سه متغیر فکر نموده و عمل فکر کردن را بین آن‌ها جایجا کند.

از آنجا که نیازی به دانستن همه آن‌ها تا زمانی که در آینده مورد استفاده قرار می‌گیرند، وجود ندارد، به راحتی می‌توان تعریف هر کدام را فقط به جایی که اولین بار استفاده خواهد شد جایجا نمود:

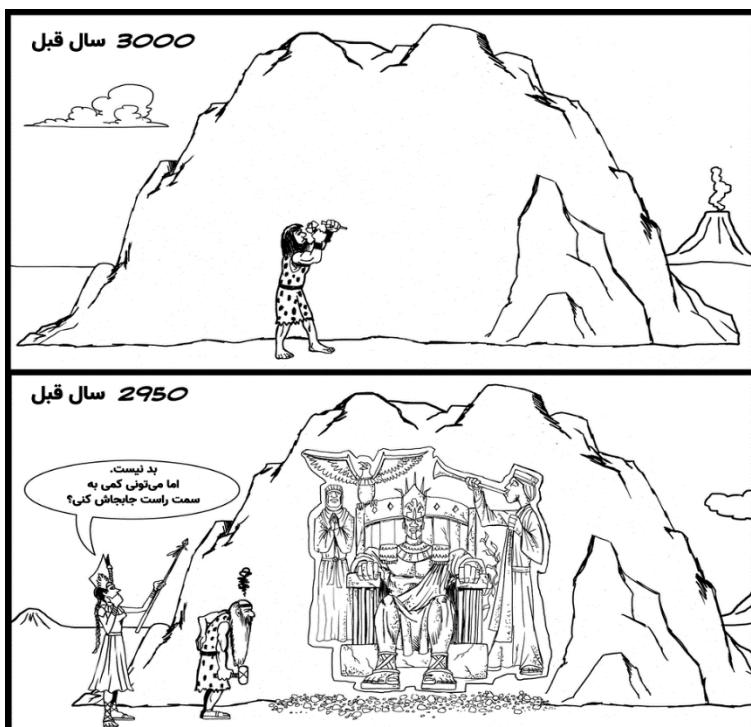
```
def ViewFilteredReplies(original_id):
    root_message = Messages.objects.get(original_id)
    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
    root_message.save()
    all_replies = Messages.objects.select(root_id=original_id)
    filtered_replies = []
    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
            filtered_replies.append(reply)
    return filtered_replies
```

شاید شما نگران باشید که `all_replies` یک متغیر ضروری است یا با انجام این کار می‌تواند حذف شود؟

```
for reply in Messages.objects.select(root_id=original_id):
    ...
```

از آن جا در این مثال `all_replies` متغیر را به خوبی توضیح می‌دهد، بنابراین ما تصمیم گرفتیم که آن را نگه داریم.

ترجیح دادن متغیرهای Write-Once^۱



تا اینجای این فصل، سخن درباره چگونگی ساخت بودن درک برنامه‌هایی بود که تعداد زیادی متغیر دارند. بی‌شک کردن در مورد متغیرهایی که دائماً در حال تغییر هستند سخت‌تر بوده و پیگیری مقدار آن‌ها دشواری دیگری را اضافه می‌کند. برای مقابله با این مشکل، ما پیشنهادی داریم که ممکن است کمی عجیب به نظر آید: متغیرهایی را که به صورت Write-Once نوشته

^۱ منظور متغیرهایی است که یک مرتبه تعریف شده و مقدار آن در طول برنامه تغییر داده نمی‌شود.

می‌شوند، ترجیح دهید چرا که راحت‌تر می‌توان در مورد اینگونه متغیرهای که یک چیز ثابت هستند فکر کرد. همچون ثابت‌ها:

```
static const int NUM_THREADS = 10;
```

استفاده از `const` در C++ (و Java نهایی) بسیار توصیه شده است.

در بسیاری از زبان‌ها (از جمله Python و Java) برخی از نوع‌های **داخلی**^۱ مانند `string` تغییر ناپذیر^۲ هستند. همان‌گونه که James Gosling (خالق جاوا) گفته است: «متغیرهای تغییرناپذیر معمولاً بدون دردرس هستند».

به یاد داشته باشید که اگر نمی‌توانید متغیر خود را به صورت Write-Once بنویسید، اگر کاری کنید که متغیر در مکان‌های کمتری تغییر کند، باز هم کمک کننده خواهد بود.

کلید طلایی

هرچه مکان یک متغیر بیشتر دستکاری شود، استدلال در مورد مقدار فعلی آن سخت‌تر خواهد شد.

شما چگونه این کار را انجام می‌دهید؟ چطور می‌توانید یک متغیر را که به صورت Write-Once است تغییر دهید؟ در اکثر مواقع، این کار نیاز به کمی بازسازی^۳ کد دارد که در مثال بعدی به آن خواهیم پرداخت.

مثال پایانی

به عنوان مثال آخر در این فصل، ما می‌خواهیم مثالی از بیشتر اصولی که تا کنون در مورد آن‌ها بحث کرده‌ایم را نشان دهیم.

فرض کنید یک صفحه وب با تعدادی فیلد متغیر `input` دارید که به شکل زیر تنظیم شده است:

```
<input type="text" id="input1" value="Dustin">
<input type="text" id="input2" value="Trevor">
<input type="text" id="input3" value="">
<input type="text" id="input4" value="Melissa">
...

```

همان‌گونه قابل مشاهده است، `id`ها با `input1` شروع شده و حالت افزایشی دارد.

^۱ built-in

^۲ immutable

^۳ restructuring

وظیفه شما نوشتن تابعی به نام `setFirstEmptyInput()` است که یک رشته را گرفته و آن را در اولین `<input>` خالی روی صفحه (که در این مثال `input3` می‌باشد) قرار دهد. این تابع یا باید المان DOM که به روزرسانی شده بود و یا مقدار `null` را اگر هیچ `input` خالی وجود نداشته باشد برگرداند. در اینجا چند کد برای انجام این کار وجود دارد که اصول مطرح شده در این فصل را رعایت نمی‌کنند:

```
var setFirstEmptyInput = function (new_value) {
    var found = false;
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            found = true;
            break;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    if (found) elem.value = new_value;
    return elem;
};
```

هر چند این کد کار خواسته شده را انجام می‌دهد، اما زیبا نیست. مشکل این کد چیست؟ و چگونه می‌توانیم آن را بهبود دهیم؟

راههای زیادی برای فکر کردن در مورد بهبود این کد وجود دارد، اما ما می‌خواهیم آن را از لحاظ متغیرهای مورد استفاده، در نظر بگیریم، یعنی:

- `var found` متغیر
- `var i` متغیر
- `var elem` متغیر

هر سه متغیر در کل تابع وجود داشته و چندین بار نوشته شده‌اند. باید سعی کنیم استفاده از هر کدام را بهبود دهیم. همانطور که در ابتدای فصل بحث کردیم، اغلب می‌توان متغیرهای

واسطه‌ای^۱ مانند found را با برگرداندن^۲ زودهنگام حذف نمود. نسخه بهبود یافته کد به این صورت است:

```
var setFirstEmptyInput = function (new_value) {
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    return null;
};
```

حال نگاهی به elem بیندازید. این متغیر چندین بار در طول کد به صورت تکراری استفاده شده است و به همین دلیل ردیابی مقدارش سخت شده است. به نظر می‌رسد که کد را از طریق مقدار تکرار می‌کنیم، در حالی که واقعاً فقط از طریق افزایش انجام می‌شود. بنابراین بایاید حلقه را در یک حلقه روی^۳ بازنویسی کنیم:

```
var setFirstEmptyInput = function (new_value) {
    for (var i = 1; true; i++) {
        var elem = document.getElementById('input' + i);
        if (elem === null)
            return null; // Search Failed. No empty input found.
        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
    }
};
```

^۱ intermediate

^۲ returning

به این مورد توجه کنید که elem چگونه به عنوان یک متغیر write-once که طول عمر آن، درون حلقه است، عمل می‌کند. استفاده از true به عنوان شرط حلقه غیرمعمول است، اما در عوض، می‌توانیم تعریف متغیر `n` و تغییراتش را در یک خط ببینیم. همچنین استفاده از یک حلقه while(true) سنتی نیز منطقی به نظر می‌رسد.

خلاصه فصل

این فصل درباره این موضوع بود که چگونه متغیرهای یک برنامه می‌توانند سریعاً روی هم تلباشند (زیاد شوند) و پیگیری آن‌ها سخت شود. شما می‌توانید با داشتن متغیرهای کمتر و تبدیل آن‌ها به صورت سبک تا حد ممکن کد خود را برای خواندن ساده‌تر کنید. به طور مشخص:

- از بین بردن متغیرهایی^۱، که فقط در مسیر هستند، همچون مدیریت مستقیم نتیجه^۱های واسط با از بین بردن متغیرهای واسط.
- کاهش محدوده هر متغیر، تا حد امکان با جایجا کردن هر متغیر به مکانی که بتوان در خطوط کمتری از کد، آن را مشاهده کرد. زیرا چیزی که خارج از دید باشد، خارج از ذهن نیز خواهد بود.
- ترجیح دادن متغیرهای `write-once`، یعنی متغیرهایی که فقط برای یک مرتبه تنظیم می‌شوند (`همچون const` یا `final` یا دیگر متغیرهای تغییر ناپذیر) که سبب ساده‌تر شدن خواندن کد می‌شوند.

^۱ intermediate result

بخش سوم

سازماندهی مجدد کد

در بخش دوم، درباره چگونگی تغییر حلقه‌ها و منطق یک برنامه صحبت کردیم تا کد شما خوانایی بیشتری پیدا کند. برای این کار چندین تکنیک تغییر ساختار برنامه، با روش‌های جزئی را شرح دادیم. در این بخش در مورد تغییرات بزرگتری که می‌توانید در سطح تابع، در کد خود اعمال کنید، بحث خواهیم کرد. برای این منظور سه روش را به طور خاص برای سازماندهی مجدد کد، پوشش خواهیم داد:

- زیرمسئله‌هایی که به هدف اصلی برنامه مربوط نمی‌شوند را استخراج کنید.
- کد خود را مجددا بازسازی کنید، بنابراین کد شما در یک لحظه فقط یک کار را انجام می‌دهد.
- ابتدا کد خود را در کلمات توصیف کنید و از این توصیفات برای راهنمایی خود به سمت یک راه حل واضح استفاده کنید.

در نهایت نیز در مورد موقعیت‌هایی صحبت خواهیم کرد که می‌توانید کد را به صورت کامل حذف کرده و یا از نوشت آن در وهله اول خودداری کنید(این راه بهترین روش برای ساده‌تر کردن کد جهت خواندن است).

فصل دهم

استخراج زیرمسئله‌های^۱ غیر مرتبط



^۱ Subproblems

مهندسی یعنی شکستن مسئله‌های بزرگ به مسائل کوچک‌تر و قرار دادن راه حل‌های این مسائل، کثار هم. اعمال این اصل در کد، آن را منسجم‌تر و برای خواندن ساده‌تر می‌کند.

توصیه این فصل، تلاش برای شناسایی و استخراج زیرمسئله‌های غیرمرتبط است. منظور ما این است که:

۱. به تابع یا بلوک کد نگاه کرده و از خود بپرسید هدف سطح بالای^۱ این کد چیست؟
۲. برای هر خط از کد، سوال کنید که آیا این خط مستقیماً برای رسیدن به آن هدف کار می‌کند؟ و یا اینکه برای حل یک زیرمسئله نامرتبط نیاز است که آن را در کد داشته باشیم؟
۳. اگر خطوط کافی، یک زیرمسئله غیرمرتبط را حل می‌کنند، آن کد را استخراج و به چند تابع جداگانه تبدیل کنید.

هرچند استخراج و تبدیل کد به چندین تابع جداگانه، احتمالاً کار هر روز شما است، اما برای این فصل، ما روی موارد خاص از استخراج زیرمسئله‌های غیرمرتبط تمرکز می‌کنیم، یعنی مواردی که کد استخراج شده اطلاعاتی از چرایی فراخوان شدنش ندارد.

همان گونه که مشاهده خواهید کرد، هر چند در عمل این یک تکنیک ساده است اما می‌تواند کد شما را به میزان قابل توجهی بهبود دهد. ترفند اصلی این است که فعالانه، به دنبال زیرمسئله‌های غیرمرتبط بگردیم. با این وجود، به دلایلی، بسیاری از برنامه‌نویسان به اندازه کافی از این تکنیک استفاده نمی‌کنند.

در این فصل به مثال‌های مختلفی خواهیم پرداخت که اجرای این تکنیک را در شرایط مختلفی که ممکن است در آن‌ها قرار بگیرید، نشان می‌دهد.

مثال مقدماتی^۲:

هدف سطح بالای کد JavaScript زیر، پیدا کردن نزدیک‌ترین مکان به نقطه داده شده است(برای اینکه با هندسه پیشرفته دچار سردرگمی نشوید، کد مربوط به آن را با به شکل italic و bold نشان داده‌ایم):

¹ high-level

² Introductory

```
// Return which element of 'array' is closest to the given latitude/longitude.
// Models the Earth as a perfect sphere.

var findClosestLocation = function (lat, lng, array) {
    var closest;
    var closest_dist = Number.MAX_VALUE;
    for (var i = 0; i < array.length; i += 1) {
        // Convert both points to radians.
        var Lat_rad = radians(lat);
        var Lng_rad = radians(lng);
        var Lat2_rad = radians(array[i].latitude);
        var Lng2_rad = radians(array[i].longitude);
        // Use the "Spherical Law of Cosines" formula.
        var dist = Math.acos(Math.sin(Lat_rad) * Math.sin(Lat2_rad) +
            Math.cos(Lat_rad) * Math.cos(Lat2_rad) *
            Math.cos(Lng2_rad - Lng_rad));
        if (dist < closest_dist) {
            closest = array[i];
            closest_dist = dist;
        }
    }
    return closest;
};
```

کد داخل حلقه درباره زیرمسئله غیرمرتبط بوده و به فاصله کروی^۱ بین دو نقطه lat/long را محاسبه می‌کند. از آنجا که تعداد زیادی کد در این قسمت وجود دارد، منطقی است که آن را به صورت یک تابع جداگانه به نام spherical_distance() استخراج کنیم:

```
var spherical_distance = function (lat1, lng1, lat2, lng2) {
    var lat1_rad = radians(lat1);
    var lng1_rad = radians(lng1);
    var lat2_rad = radians(lat2);
    var lng2_rad = radians(lng2);
    // Use the "Spherical Law of Cosines" formula.
    return Math.acos(Math.sin(lat1_rad) * Math.sin(lat2_rad) +
        Math.cos(lat1_rad) * Math.cos(lat2_rad) *
        Math.cos(lng2_rad - lng1_rad));
```

^۱ spherical

};

حال باقیمانده کد به این شکل می‌شود:

```
var findClosestLocation = function (lat, lng, array) {
    var closest;
    var closest_dist = Number.MAX_VALUE;
    for (var i = 0; i < array.length; i += 1) {
        var dist = spherical_distance(lat, lng, array[i].latitude, array[i].longitude);
        if (dist < closest_dist) {
            closest = array[i];
            closest_dist = dist;
        }
    }
    return closest;
};
```

این کد خوانایی بیشتری دارد زیرا خواننده می‌تواند بدون اینکه حواسش به معادلات هندسی^۱ پرتاب شود، بر روی هدف سطح بالای کد تمرکز کند.

امتیاز دیگر این کد این است که `spherical_distance()` برای تست ساده‌تر می‌باشد. همچنین `spherical_distance()` تابعی است که می‌تواند در آینده دوباره مورد استفاده قرار گیرد و به همین دلیل یک زیرمسئله غیرمرتبط است. این تابع کاملاً خود-مختار^۲ بوده و از نحوه استفاده برنامه‌ها از خود آگاهی ندارد.

کدهایی با کاربرد خاص^۳

مجموعه‌های از ابزارهای پایه وجود دارد که اکثر برنامه‌ها از آن‌ها استفاده می‌کنند، مانند دستکاری رشته‌ها، استفاده از جداول هش^۴ و خواندن و نوشتن روی یک فایل.

اغلب این «ابزارهای پایه» توسط کتابخانه‌های داخلی(built-in) به زبان برنامه‌نویسی شما پیاده سازی شده‌اند. بعونه اگر بخواهید در PHP محتوای داخل یک فایل را بخوانید، می‌توانید تابع `file_get_contents("filename")` را فراخوانی کنید و یا در زبان Python می‌توانید از تابع `open("filename").read()` استفاده کنید. اما زمان‌هایی نیز هست که مجبورید خودتان دست به

^۱ geometry

^۲ self-contained

^۳ Pure Utility Code

^۴ hash tables

کار شده و برخی کمیودها را برطرف کنید. به عنوان مثال در C++ هیچ روش مختصراً برای خواندن کل یک فایل وجود ندارد. بنابراین ناگزیر هستید که کدی شبیه این کد را بنویسید:

```
ifstream file(file_name);
// Calculate the file's size, and allocate a buffer of that size.
file.seekg(0, ios::end);
const int file_size = file.tellg();
char* file_buf = new char [file_size];
// Read the entire file into the buffer.
file.seekg(0, ios::beg);
file.read(file_buf, file_size);
file.close();
...
```

این یک مثال سنتی از یک زیرمسئله غیرمرتبط است که باید آن را در یک تابع جدید مثل ReadFileToString() استخراج کنید. اکنون بقیه کدپایه شما می‌توانند همانند زمانی عمل کند که انگار زبان C++ از قبل تابع ReadFileToString() را داشته است.

به طور کلی هر گاه در حال فکر کردن به این جمله بودید که «ای کاش کتابخانه ما یک تابع XYZ() داشت»، دست به کار شده و آن را بنویسید! رفته رفته، مجموعه‌ای از کدهای سودمند به دست خواهید آورد که می‌تواند در پروژه‌ها مختلف مورد استفاده قرار گیرند.

سایر کدهای همه منظوره^۱

هنگام اشکال‌زدایی در JavaScript اغلب از تابع alert() برای نمایش یک باکس پیام استفاده می‌شود (یعنی همان نسخه وِب تابع printf() برای اشکال‌زدایی) که برخی اطلاعات را به برنامه‌نویس نشان می‌دهد. به عنوان مثال تابع زیر داده‌های ثبت شده را با استفاده از Ajax به سرور فراخوانی می‌کند و سپس دیکشنری برگشت داده شده از سرور را نشان می‌دهد:

```
ajax_post({
  url: 'http://example.com/submit',
  data: data,
  on_success: function (response_data) {
    var str = "{\n";
    for (var key in response_data) {
      str += " " + key + " = " + response_data[key] + "\n";
    }
    str += "}";
    document.getElementById("result").innerHTML = str;
  }
});
```

^۱ General-Purpose

```

        }
        alert(str + "}");
    }

    // Continue handling 'response_data' ...
}

});

```

هدف سطح بالای این کد این است که یک فراخوانی Ajax به سرور ایجاد کند و پاسخ آن را مدیریت کند. اما بسیاری از کدها در حال حل زیرمسئله غیرمرتبه برای چاپ زیبای^۱ یک دیکشنری هستند. استخراج این کد در یک تابع جدید مانند format_pretty(obj) کاری ساده است:

```

var format_pretty = function (obj) {
    var str = "{\n";
    for (var key in obj) {
        str += "    " + key + " = " + obj[key] + "\n";
    }
    return str + "}";
};

```

مزایای غیرمنتظره

دلایل زیادی وجود دارد که چرا استخراج تابع format_pretty() ایده خوبی است. این فراخوانی کد را ساده‌تر کرده و تابع format_pretty() را برای استفاده در دیگر مکان‌ها نیز مفید می‌کند.

دلیل مهم‌تری نیز وجود دارد که به وضوح آشکار نیست: بهبود format_pretty() هنگامی که به صورت یک تابع تکی است، کار ساده‌تری است. وقتی که شما در یک محیط ایزوله روی یک تابع کوچک‌تر کار می‌کنید، اضافه کردن ویژگی‌ها^۲، بهبود خوانایی کد، مراقبت از موارد حاشیه‌ای و دیگر موارد، ساده‌تر به نظر می‌رسد.

در اینجا مواردی وجود دارد که format_pretty(obj) توان مدیریت آن‌ها را ندارد:

- این تابع انتظار دارد که obj یک شئ^۳ باشد. اگر به جای آن یک رشته ساده یا چیز نامشخصی باشد، کد فعلی یک استثناء^۴ ایجاد می‌کند.

^۱ Pretty-print

^۲ features

^۳ object

^۴ exception

- این تابع انتظار دارد، هر مقداری از `obj` یک نوع ساده باشد. اگر این شامل object‌های تودرتو باشد، کد تابع، آن‌ها را به صورت [Object Object] نمایش خواهد داد که خیلی زیبا نیست.

قبل از اینکه `format_pretty()` را از تابع خود تفکیک کنیم، احساس می‌شود که انجام این بهبودها نیازمند کار زیادی باشد (در واقع، چاپ object‌های تودرتو به صورت بازگشته آن هم بدون تابع جداگانه، کار بسیار دشواری است). اما بعد از تفکیک، اضافه کردن این قابلیت ساده خواهد بود. در ادامه کد بهبود یافته را مشاهده می‌کنید:

```
var format_pretty = function (obj, indent) {
  // Handle null, undefined, strings, and non-objects.
  if (obj === null) return "null";
  if (obj === undefined) return "undefined";
  if (typeof obj === "string") return '"' + obj + '"';
  if (typeof obj !== "object") return String(obj);
  if (indent === undefined) indent = "";
  // Handle (non-null) objects.
  var str = "{\n";
  for (var key in obj) {
    str += indent + " " + key + " = ";
    str += format_pretty(obj[key], indent + " ") + "\n";
  }
  return str + indent + "}";
};
```

این کد کاستی‌های ذکر شده را پوشش می‌دهد و خروجی زیر را تولید می‌کند:

```
{
  key1 = 1
  key2 = true
  key3 = undefined
  key4 = null
  key5 = {
    key5a = {
      key5a1 = "hello world"
    }
  }
}
```

ساختن کدهای همه منظوره^۱ به تعداد زیاد

توابع (format.pretty() و ReadFileToString()) مثال‌های بسیار خوبی از زیرمسئله‌های غیرمرتبه هستند. آن‌ها به راحتی قابل استخراج بوده و می‌توانند در طول پروژه‌های مختلف، مجدداً مورد استفاده قرار گیرند. کدپایه یک پروژه اغلب یک مسیر^۲ ویژه برای کدهای همه منظوره دارد (مثلاً پوشه /util) تا بتوان به راحتی آن‌ها را با دیگر پروژه‌ها به اشتراک گذاشت.

کدهای همه منظوره خیلی عالی هستند زیرا به صورت کامل از بقیه پروژه شما جدا می‌شوند. کدی مانند این، برای توسعه آسان‌تر، برای تست راحت‌تر و برای فهمیدن نیز ساده‌تر است.

به بسیاری از کتابخانه‌ها و سیستم‌های قدرتمند مورد استفاده خود، مانند دیتابیس‌های SQL، کتابخانه‌های JavaScript و سیستم‌های قالب‌بندی^۳ HTML فکر کنید. لازم نیست نگران داخل آن‌ها باشید. این کدهای پایه از پروژه شما جداسازی شده‌اند و در نتیجه، کدپایه پروژه شما کوچک باقی می‌ماند.

هرچه بیشتر پروژه خود را به عنوان کتابخانه‌های جدا تفکیک کنید، بهتر خواهد بود. زیرا بقیه کد شما کوچک‌تر شده و فکر کردن درباره آن راحت‌تر خواهد بود.

آیا این برنامه‌نویسی بالا-به-پایین^۴ است یا پایین-به-بالا^۵؟

برنامه‌نویسی بالا-به-پایین سبکی است که در آن مازول‌ها و توابع بالاترین-سطح، ابتدا طراحی می‌شوند و توابع سطح-پایین در صورت نیاز برای پشتیبانی از توابع سطح-بالا، پیاده سازی می‌شوند. برنامه‌نویسی پایین-به-بالا سعی دارد ابتدا همه زیرمسئله‌ها را پیش بینی و حل کند و سپس کامپوننت‌های سطح-بالا را با استفاده از این اجزاء بسازد. در این فصل از یک رویکرد در مقابل رویکرد دیگر دفاع نمی‌شود خصوصاً که اکثر برنامه‌نویسی‌ها ترکیبی از هر دو رویکرد است. هدف اصلی این است که زیرمسئله‌ها حذف شده و به صورت جداگانه حل شوند.

^۱ General-Purpose

^۲ directory

^۳ templateing

^۴ TOP-DOWN

^۵ BOTTOM-UP

قابلیت‌های پروژه-محور^۱

در حالت ایده‌آل، زیرمسئله‌ای که شما استخراج می‌کنید، به طور کامل project-agnostic^۲ خواهد بود. حتی اگر اینگونه نباشد، باز هم اشکالی ندارد، شکستن زیرمسئله‌ها هنوز هم باعث شگفتی می‌شود.

در اینجا مثالی از یک وبسایت بررسی کسب و کار، آورده شده است. این کد Python، ابتدا یک شعبه ایجاد می‌کند و برای آن name، url و date_created را تنظیم می‌کند:

```
business = Business()
business.name = request.POST["name"]
url_path_name = business.name.lower()
url_path_name = re.sub(r"\.", "", url_path_name)
url_path_name = re.sub(r"[^a-z0-9]+", "-", url_path_name)
url_path_name = url_path_name.strip("-")
business.url = "/biz/" + url_path_name
business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

A.C. Joe's Tire & Smog, Inc^۳ قرار است url نسخه تمیزی از name باشد. به عنوان مثال اگر name برابر /biz/ac-joes-tire-smog-inc URL معتبر است. ما می‌توانیم این کد را سریع و راحت استخراج کنیم و در حالی که در این مرحله هستیم عبارات منظم^۴ را از قبل کامپایل نموده و به آن‌ها اسامی قابل خواندن بدھیم:

```
CHARS_TO_REMOVE = re.compile(r"\.+")
CHARS_TO_DASH = re.compile(r"[^a-z0-9]+")
def make_url_friendly(text):
    text = text.lower()
    text = CHARS_TO_REMOVE.sub(' ', text)
    text = CHARS_TO_DASH.sub('-', text)
    return text.strip("-")
```

^۱ Project-Specific Functionality

^۲ به چیزی اطلاق می‌شود که بتواند تعمیم یابد یا به گونه‌ای در بین سیستم‌های مختلف قابل سازگار باشد.

^۳ regular expressions

اکنون کد اصلی الگوی «منظمری^۱» دارد:

```
business = Business()
business.name = request.POST["name"]
business.url = "/biz/" + make_url_friendly(business.name)
business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

این کد برای خوانده شدن به تلاش کمتری نیاز دارد، زیرا با وجود عبارات منظم و دستکاری رشته‌های عمیق^۲، دچار ابهام نمی‌شود.

حال این سوال مطرح است که باید کد مربوط به make_url_friendly() را کجا قرار دهید؟ از آنجا که به نظر می‌رسد یک تابع نسبتاً کلی است، بنابراین شاید قرار دادن آن در مسیر جدآگاهه util منطقی به نظر برسد اما از سوی دیگر، این عبارات منظم با نام تجاری U.S طراحی شده‌اند، بنابراین شاید این کد باید در همان فایلی که استفاده شده است باقی بماند. این واقعاً مهم نیست که حجم آن زیاد باشد، به راحتی می‌توانید تعریف آن را در آینده تغییر دهید. نکته مهم‌تر این است که make_url_friendly() به هیچ وجه استخراج نشده بود.

ساده سازی Interface موجود

همه افراد، کتابخانه‌ای که یک interface واضح و تمیز ارائه می‌دهد را دوست دارند. مواردی که آرگومان‌های کمی گرفته، تنظیمات راه اندازی زیادی نداشته و به طور کلی تلاش کمی برای استفاده از آن‌ها نیاز است. این باعث می‌شود کد شما زیبا به نظر برسد: هم زمان ساده و قدرتمند.

اما اگر interface مورد استفاده واضح نبود، هنوز هم می‌توانید توابع wrapper^۳ خود را که واضح هستند، بسازید.

^۱ regular

^۲ deep string

^۳ یک تابع wrapper یک زیروال در یک کتابخانه نرم افزار و یا یک برنامه کامپیوتري است که هدف اصلی آن فراخوانی یک زیروال دوم و یا یک فراخوان سیستمی با کمی و یا بدون محاسبات اضافی است.

به عنوان مثال، کار با کوکی‌های^۱ مرورگر در JavaScript خیلی ایده آل نیست. از نظر مفهومی، کوکی‌ها مجموعه‌ای از جفت‌های name/value هستند. اما فراهم شده توسط مرورگر یک رشته تکی syntax را ارائه می‌دهد که آن به شکل زیر است:

```
name1=value1; name2=value2; ...
```

برای پیدا کردن کوکی مورد نظر مجبور به تجزیه این رشته غول پیکر هستید. در اینجا کدی داریم که مقدار کوکی با نام max_results را می‌خواند:

```
var max_results;
var cookies = document.cookie.split(';');
for (var i = 0; i < cookies.length; i++) {
    var c = cookies[i];
    c = c.replace(/^\s+/); // remove leading spaces
    if (c.indexOf("max_results=") === 0)
        max_results = Number(c.substring(12, c.length));
}
```

ظاهرا این کد خیلی رشت به نظر می‌رسد. همانطور که انتظار می‌رود، تابع() از قبل نوشته شده است، بنابراین ما می‌توانیم فقط بنویسیم:

```
var max_results = Number(get_cookie("max_results"));
```

عجیب‌تر از این، ساختن یا تغییر دادن مقدار یک کوکی است. شما مجبورید document.cookie را با یک مقدار دقیقاً به شکل زیر تنظیم کنید:

```
document.cookie = "max_results=50; expires=Wed, 1 Jan 2020 20:53:47 UTC; path=/";
```

ظاهرا باید این دستور تمام کوکی‌های موجود دیگر را نیز بازنویسی کند، اما این کار را انجام نمی‌دهد.

یک interface ایده‌آل‌تر برای تنظیم کوکی چیزی شبیه این خواهد بود:

```
set_cookie(name, value, days_to_expire);
```

^۱ cookie

پاک کردن یک کوکی همچنان کاری غیر معمول است چرا که باید از قبل برای آن زمان انقضای تعیین کنید. در عوض یک interface ایده‌آل به سادگی به صورت زیر خواهد بود:

```
delete_cookie(name);
```

موضوع اصلی اینجا این است که **شما هرگز نباید به یک interface غیر ایده‌آل راضی شوید.** شما همواره می‌توانید توابع wrapper خود را برای مخفی کردن جزئیات زشت interface‌هایی که در آن گیر افتاده‌اید، ایجاد کنید.

تغییر شکل^۱ مجدد Interface با توجه به نیاز

کدهای زیادی در یک برنامه فقط برای پشتیبانی از کدهای دیگر استفاده می‌شوند همچون تنظیم ورودی‌ها برای یک تابع یا ارسال خروجی‌های پردازش شده. این کد «چسبان^۲» معمولاً هیچ ارتباطی با منطق اصلی برنامه شما ندارد. کدهای بدون تغییری مانند این، گزینه‌ای عالی برای دریافت^۳ از توابع، بصورت جداگانه هستند.

به عنوان مثال، فرض کنید یک دیکشنری Python که شامل اطلاعات حساس کاربر مانند {"username": "...", "password": "..."} داشته و باید همه این اطلاعات را در URL قرار دهید. به دلیل حساس بودن این اطلاعات، تصمیم می‌گیرید که ابتدا دیکشنری را با استفاده از یک کلاس Cipher رمزگاری کنید.

اما باید توجه کنید که Cipher از یک سو انتظار دارد که رشته‌ای از بایت‌های^۴(نه یک دیکشنری) را به عنوان ورودی دریافت کند و از سوی دیگر نیز یک رشته از بایت‌ها را بر می‌گرداند، اما چیزی که ما نیاز داریم یک URL امن است. Cipher همچنین تعدادی پارامتر اضافی را گرفته و برای استفاده خیلی دست و پا گیر است.

آنچه به عنوان یک کار ساده شروع شده بود به کد چسبان^۵ بزرگ تبدیل می‌شود:

```
user_info = { "username": "...", "password": "..." }
user_str = json.dumps(user_info)
cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
encrypted_bytes = cipher.update(user_str)
```

^۱ Reshaping

^۲ glue

^۳ pull

^۴ string of bytes

^۵ glue code

```
encrypted_bytes += cipher.final() # flush out the current 128 bit block
url = "http://example.com/?user_info=" + base64.urlsafe_b64encode(encrypted_bytes)
...
```

حتی اگر بتوانیم مشکل را با رمزگاری اطلاعات کاربر داخل یک URL حل کنیم، باز قسمت اعظم کد فقط در حال رمزگاری این شئ Python در یک رشته URL-frendly است. راه حل ساده استخراج زیرمسئله است:

```
def url_safe_encrypt(obj):
    obj_str = json.dumps(obj)
    cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
    encrypted_bytes = cipher.update(obj_str)
    encrypted_bytes += cipher.final() # flush out the current 128 bit block
    return base64.urlsafe_b64encode(encrypted_bytes)
```

حال نتیجه کد برای اجرای منطق واقعی برنامه، ساده می‌شود:

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + url_safe_encrypt(user_info)
```

دور نگه داشتن بیش از حد توابع از یکدیگر

همان گونه که در ابتدای این فصل گفتیم، هدف ما پشتکار^۱ بیشتر برای شناسایی و استخراج زیرمسئله‌های غیرمرتبط است. ما می‌گوییم پشتکار، زیرا اکثر کدنویسان به اندازه کافی پرتلاش نیستند. اما گاهی این امکان وجود دارد که چیزها را بیش از حد جداسازی کنید.

به عنوان مثال، کد بخش قبلی می‌توانست به موارد بیشتری مانند کد زیر شکسته شود:

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + url_safe_encrypt_obj(user_info)
def url_safe_encrypt_obj(obj):
    obj_str = json.dumps(obj)
    return url_safe_encrypt_str(obj_str)
def url_safe_encrypt_str(data):
    encrypted_bytes = encrypt(data)
    return base64.urlsafe_b64encode(encrypted_bytes)
def encrypt(data):
```

^۱ aggressive

```

cipher = make_cipher()
encrypted_bytes = cipher.update(data)
encrypted_bytes += cipher.final() # flush out any remaining bytes
return encrypted_bytes
def make_cipher():
    return Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)

```

بی شک معرفی همه این توابع کوچک، به خوانایی کد آسیب می‌زند، زیرا خواننده باید حضور ذهن بیشتری داشته باشد، خصوصاً که دنبال کردن مسیر اجرایی، نیازمند پرش به اطراف است. ولی به هر حال اضافه کردن توابع جدید شامل یک هزینه خوانایی کوچک(اما ملموس) است که باید بپردازید و اندکی از خوانایی کد صرف نظر کنید.

خلاصه فصل

هدف اصلی این فصل ارائه روشی ساده یعنی جدا کردن کد عمومی از کدهای ویژه پروژه است. از آنجا که بیشتر قسمت‌های کد، عمومی است، برای حل مشکلات کلی می‌توان یک مجموعه بزرگ از کتابخانه‌ها و توابع کمکی را ایجاد نمود و آنچه که باقی می‌ماند یک هسته کوچک خواهد بود که برنامه شما را منحصر به فرد می‌نماید.

دلیل اصلی مفید بودن این تکنیک این است که به برنامه‌نویسان اجازه می‌دهد تا روی مشکلات کوچکتر که به خوبی تعریف شده و از بقیه پروژه جدا شده‌اند تمرکز کنند. در نتیجه، راه حل‌های بهتر و صحیح‌تری برای این زیرمسئله‌ها پیدا خواهید شد. مزیت آخر نیز اینکه امکان استفاده مجدد از آن‌ها در آینده وجود دارد.

پیشنهاد مطالعه بیشتر

در کتاب Refactoring از Martin Fowler روشی برای بهبود طراحی کد موجود در «متد استخراجی» با عنوان بازسازی توصیف شده است و همچنین بسیاری از روش‌های دیگر، برای بازسازی کد در این کتاب ارائه شده است.

تعدادی از اصول مربوط به شکستن کد به توابع کوچک است. به خصوص، یکی از اصل‌ها این است: همه عملیات‌ها^۱ را در یک متد تکی در همان سطح انتزاع^۲ نگهداری کنید.

این ایده‌ها، مشابه توصیه ما در مورد استخراج زیرمسئله‌های غیرمرتبط است. آنچه در این فصل مورد بحث قرار دادیم، یک مورد ساده و خاص، از استخراج یک متد است.

^۱ operations

^۲ abstraction

فصل یازدهم

در هر لحظه یک وظیفه^۱



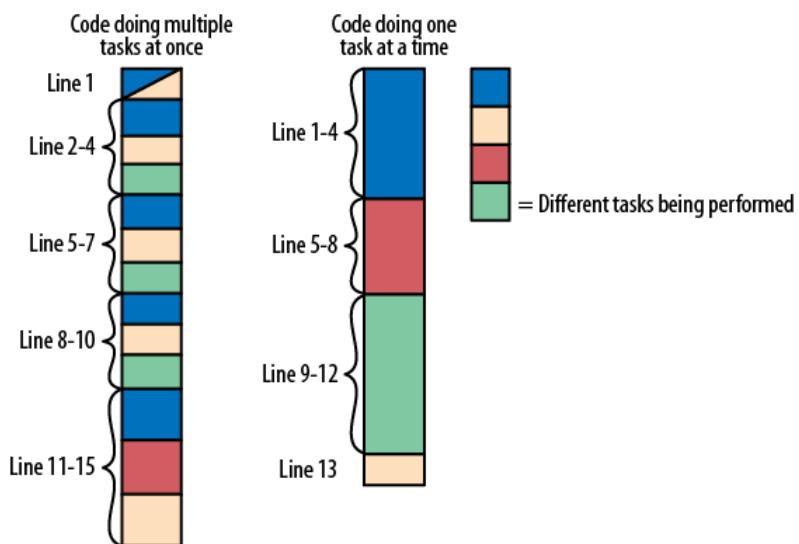
Task

درک کردن کدی که چندین کار را به شکل همزمان انجام می‌دهد، سخت‌تر است. یک بلوک تکی کد ممکن است اشیای جدید^۱، پالایش داده^۲، تجزیه و رویدی‌ها^۳ و اعمال منطق کسب و کار را به صورت هم زمان آماده سازی^۴ کند. درک کل این کدهای در کنار هم بافته شده نسبت به زمانی که هر وظیفه^۵ به تنها یک شروع و تکمیل شود، سخت‌تر است.

کلید طلایی

کد باید به گونه‌ای سازماندهی شده باشد که در یک لحظه فقط یک وظیفه را انجام دهد.

به بیان دیگر، این فصل درباره «یکپارچه‌سازی^۶» کدهای شما است. نمودار زیر، این فرآیند را نشان می‌دهد. سمت چپ تصویر وظایف مختلف انجام شده توسط یک قطعه کد را نشان داده و سمت راست تصویر، همان کد را، بعد از سازماندهی آن برای انجام هر وظیفه در یک لحظه، نشان می‌دهد.



شاید این توصیه را شنیده باشید که: توابع فقط باید یک کار را انجام دهند. توصیه ما نیز شبیه همین است، چرا که همیشه درباره مرزهای^۷ تابع صدق نمی‌کند. مطمئناً شکستن یک تابع بزرگ

^۱ new objects

^۲ cleansing data

^۳ parsing inputs

^۴ initializing

^۵ task

^۶ defragmenting

^۷ boundaries

به چندین تابع کوچک می‌تواند خوب باشد. اما اگر این کار را هم نکنید، هنوز می‌توانید کد را در داخل همان تابع بزرگ نیز سازمان دهی کرده و این احساس را ایجاد کنید که بخش‌های منطقی، از هم جدا شده‌اند.

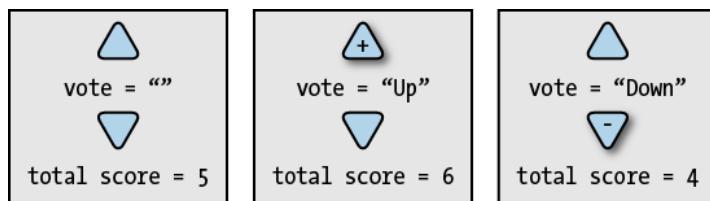
فرآیند زیر کاری است که ما برای ساختن «کدی که در یک لحظه فقط یک وظیفه را انجام می‌دهد» استفاده می‌کنیم:

۱. تمام وظایفی که کد شما انجام می‌دهد را لیست کنید. منظور ما از وظیفه (task) معنایی بسیار گسترده است که می‌تواند به کوچکی این جمله: «اطمینان حاصل کنید که این object معتبر است» یا به معنی این جمله: «از طریق تکرار هر گره ادر درخت» باشد.
۲. سعی کنید این وظایف را تا حد ممکن در توابع مختلف جداسازی کرده و حداقل در بخش‌های مختلف کد تقسیم کنید.

وظیفه‌ها می‌توانند کوچک باشند

فرض کنید یک ابزارک رای گیری در یک وبلاگ وجود دارد تا کاربر بتواند به یک کامنت، امتیاز مثبت (Up) یا منفی (Down) بدهد. مجموع امتیاز برای یک کامنت به این شکل محاسبه می‌شود که برای هر رای مثبت +1 و برای هر رای منفی -1 امتیاز تعلق می‌گیرد.

در اینجا سه حالت برای یک کاربر و اینکه چگونه می‌تواند بر امتیاز کل اثر بگذارد وجود دارد:



زمانی که کاربر روی دکمه کلیک می‌کند (برای رای دادن یا تغییر رای) کد زیر صدا زده می‌شود:

```
vote_changed(old_vote, new_vote); // each vote is "Up", "Down", or ""
```

این تابع مجموع امتیاز را به روزرسانی می‌کند و برای همه ترکیب‌های old_vote/new_vote اجرا می‌شود:

¹ node

```

var vote_changed = function (old_vote, new_vote) {
  var score = get_score();
  if (new_vote !== old_vote) {
    if (new_vote === 'Up') {
      score += (old_vote === 'Down' ? 2 : 1);
    } else if (new_vote === 'Down') {
      score -= (old_vote === 'Up' ? 2 : 1);
    } else if (new_vote === '') {
      score += (old_vote === 'Up' ? -1 : 1);
    }
  }
  set_score(score);
};

```

اگرچه کد از نظر کوتاهی خیلی خوب است، اما کارهای زیادی را انجام می‌دهد. جزئیات پیچدهای وجود داشته و این دشوار است که با یک نگاه اجمالی بگویید، آیا خطاهای غیرمتربقه، خطاهای تایپوگرافی^۱ یا باگ‌های دیگر وجود دارد یا نه؟ ظاهرا این کد تنها یک کار (یعنی بهروزرسانی امتیاز) را انجام می‌دهد اما در واقع دو وظیفه وجود دارد که به صورت هم‌زمان در حال انجام است:

.۱. new_vote و old_vote به مقادیر عددی تبدیل^۲ می‌شوند.

.۲. امتیاز بهروزرسانی می‌شود.

ما می‌توانیم با انجام هر وظیفه به صورت جداگانه، خواندن کد را آسان‌تر کنیم. کد زیر اولین وظیفه برای تبدیل رای به یک مقدار عددی را حل می‌کند:

```

var vote_value = function (vote) {
  if (vote === 'Up') {
    return +1;
  }
  if (vote === 'Down') {
    return -1;
  }
  return 0;
};

```

^۱ typos

^۲ parse

```
};
```

حال بقیه کد می‌تواند وظیفه دوم، یعنی به روزرسانی امتیاز را حل کند:

```
var vote_changed = function (old_vote, new_vote) {
  var score = get_score();
  score -= vote_value(old_vote); // remove the old vote
  score += vote_value(new_vote); // add the new vote
  set_score(score);
};
```

همان گونه که می‌بینید: این نسخه از کد، تلاش ذهنی کمتری برای متقاءعد کردن شما در این مورد که این کد کار می‌کند، نیاز دارد. این نکته‌ای مهم در مورد دلایل آسان شدن درک کد است.

استخراج مقدارها از یک Object

زمانی ما یک کد JavaScript داشتیم که مکان یک کاربر را در یک رشته از City, Country می‌داد. مانند Paris, France یا Santa Monica, USA. در واقع ما یک دیکشنری location_info با اطلاعات ساخت‌یافته فراوان داشتیم. تنها کاری که باید انجام می‌دادیم این بود که، یک شهر و یک کشور را از همه فیلدّها انتخاب و سپس آن‌ها را به هم الحاق کنیم. تصویر زیر مثالی از ورودی و خروجی این کد را نشان می‌دهد:

location_info	
LocalityName	"Santa Monica"
SubAdministrativeAreaName	"Los Angeles"
AdministrativeAreaName	"California"
CountryName	"USA"

↓

"Santa Monica, USA"

ابتدا به نظر می‌رسد که با چیز ساده‌ای روبرو هستیم، اما قسمت مشکل این است که برخی یا همه این چهار مقدار، ممکن است وجود نداشته باشند. در اینجا نحوه برخورد با آن‌ها آورده شده است:

- هنگام انتخاب شهر، ما ترجیح می‌دادیم در صورت وجود، ابتدا از LocalityName که همان larger (city/town) است، استفاده کنیم و سپس از SubAdministrativeAreaName یا همان (state/territory) و بعد از آن از AdministrativeAreaName یا همان (city/county) استفاده کنیم.
- اگر هر سه مورد وجود نداشته باشند، نام شهر، با توجه به مقدار پیش‌فرض MiddleofNowhere تعیین می‌شد.
- اگر نام CountryName وجود نداشته باشد، عبارت Planet Earth به عنوان پیش‌فرض انتخاب می‌شد.

تصویر زیر دو نمونه از مدیریت مقدارهای از دست رفته یا ناموجود را نشان می‌دهد:

location_info	
LocalityName	(undefined)
SubAdministrativeAreaName	(undefined)
AdministrativeAreaName	(undefined)
CountryName	"Canada"

↓

"Middle-of-Nowhere, Canada"

location_info	
LocalityName	(undefined)
SubAdministrativeAreaName	"Washington, DC"
AdministrativeAreaName	(undefined)
CountryName	"USA"

↓

"Washington, DC, USA"

در ادامه کدی را که برای پیاده سازی این وظیفه نوشته‌ایم، آورده شده است:

```
var place = location_info["LocalityName"]; // e.g. "Santa Monica"
if (!place) {
    place = location_info["SubAdministrativeAreaName"]; // e.g. "Los Angeles"
}
if (!place) {
    place = location_info["AdministrativeAreaName"]; // e.g. "California"
}
if (!place) {
    place = "Middle-of-Nowhere";
}
if (location_info["CountryName"]) {
    place += ", " + location_info["CountryName"]; // e.g. "USA"
} else {
    place += ", Planet Earth";
}
return place;
```

طمئناً این کمی کثیف است اما کار مد نظر ما را انجام می‌داد.

چند روز بعد ما نیازمند بهبود این عملکرد شدیم چرا که می‌خواستیم برای مکان‌هایی در United States، به جای نام کشور (county) نام ایالت (state) را در صورت وجود، نمایش دهیم و در نتیجه به جای Santa Monica، California مقدار Santa Monica، USA بازگردانده می‌شد. بی‌شک افزودن این ویژگی به کد قبلی سبب زشت‌تر شدن آن خواهد شد.

اعمال کردن «یک وظیفه در یک لحظه»

به جای اعمال فشار برای تغییر این کد به چیزی که می‌خواستیم، کمی درنگ کرده و متوجه شدیم که این کد، قبل از چندین وظیفه را به طور همزمان انجام می‌داده است:

۱. استخراج مقدارها از دیکشنری location_info
۲. City را از طریق ترتیب اولویت به دست آورید، در صورت پیدا نکردن چیزی، مقدار پیش‌فرض را برابر Middle-of-Nowhere قرار دهید.
۳. به دست آوردن Country و در صورتی که وجود نداشت از مقدار Planet Earth استفاده کنید.
۴. مکان را به روزرسانی کنید

نتیجه این که، ما کد اصلی را بازنویسی خواهیم کرد تا هر یک از این وظیفه‌ها به صورت مستقل حل شوند. اولین وظیفه یعنی استخراج مقدارها از location_info به راحتی قابل حل است:

```
var town    = location_info["LocalityName"];           // e.g. "Santa Monica"
var city    = location_info["SubAdministrativeAreaName"]; // e.g. "Los Angeles"
var state   = location_info["AdministrativeAreaName"]; // e.g. "CA"
var country = location_info["CountryName"];           // e.g. "USA"
```

در این مرحله، کار ما با استفاده از location_info انجام شده است و لازم نیست آن کلیدهایی طولانی را به خاطر بسپاریم. در عوض، چهار متغیر ساده برای کار با آن‌ها داریم. در مرحله بعد، باید بفهمیم که مقدار برگشته second_half چه چیزی باید باشد:

```
// Start with the default, and keep overwriting with the most specific value.
var second_half = "Planet Earth";
if (country) {
  second_half = country;
}
if (state && country === "USA") {
```

^۱ keys

```
second_half = state;
}
```

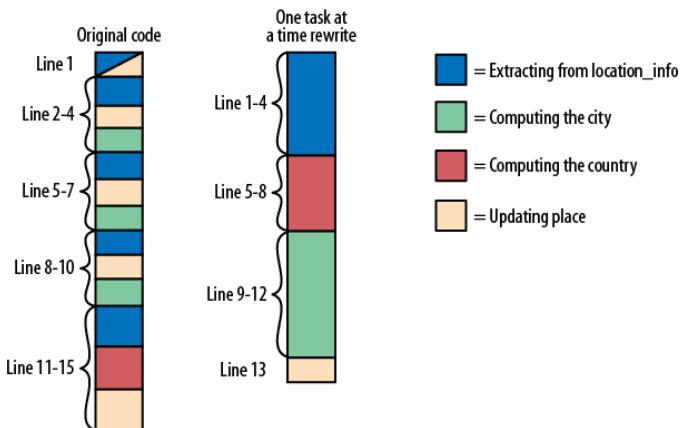
به طور مشابه، می‌توانیم مقدار first_half را نیز بدانیم:

```
var first_half = "Middle-of-Nowhere";
if (state && country !== "USA") {
    first_half = state;
}
if (city) {
    first_half = city;
}
if (town) {
    first_half = town;
}
```

و در نهایت ما اطلاعات را به یکدیگر می‌چسبانیم:

```
return first_half + ", " + second_half;
```

تصویر یکپارچه سازی^۱ در ابتدای این فصل، در واقع یک نمایش مجدد از راه حل اصلی و نسخه جدید بود. در اینجا همان تصویر با جزئیات بیشتر ارائه شده است:



همان گونه که می‌بینید، در راه حل دوم چهار وظیفه به مناطق مجزا تقسیم شده است.

^۱ Fragmentation

یک رویکرد دیگر

هنگام بازسازی^۱ کد، اغلب چندین راه وجود دارد و این مورد نیز از این قاعده مستثنی نیست.

هنگامی که برخی از این وظایف را جدا کردید، فکر کردن در مورد کد راحت‌تر شده و ممکن است روش‌های بهتری برای بازسازی مجدد پیدا کنید. به عنوان نمونه، برای درک کردن مجموعه دستورات if اخیر، به دقت بیشتری برای خواندن کد نیاز داریم که آیا هر مورد به درستی کار می‌کند یا نه؟ در واقع دو زیروظیفه^۲ به صورت همزمان در کد وجود دارد:

لیستی از متغیرها را مرور کرده و در صورت وجود یکی را که ارجحیت بیشتری دارد، انتخاب کنید.

بسته به اینکه کشور USA باشد، یک لیست متفاوت انتخاب کنید.

با نگاه دوباره به قبل، می‌توانید بینید که کد دارای منطق «if USA» با بقیه منطق کد، به هم آمیخته شده است. در عوض، می‌توانیم موارد USA و غیر USA را به صورت جداگانه مدیریت کنیم:

```
var first_half, second_half;
if (country === "USA") {
    first_half = town || city || "Middle-of-Nowhere";
    second_half = state || "USA";
} else {
    first_half = town || city || state || "Middle-of-Nowhere";
    second_half = country || "Planet Earth";
}
return first_half + ", " + second_half;
```

در صورتی که با JavaScript آشنا نیستید، باید بدانید که عبارت a || b || c یک عبارت اتمیک است و با اولین مقدار true، ارزیابی خاتمه می‌یابد (در این مورد، رشته تعریف شده، یک رشته تهی^۳ نیست). مزیتی که این کد دارد این است که، بررسی کردن لیست‌های برگزیده و بهروزرسانی آن‌ها را بسیار ساده کرده است. همچنین بیشتر دستورات if حذف شده‌اند و منطق تجاری مجدداً توسط خطوط کمتری، پیاده‌سازی شده است.

^۱ Refactoring

^۲ subtasks

^۳ nonempty

یک مثال بزرگتر

در یک سیستم خزیدن در وب^۱ که ساخته بودیم، یک تابع با نام UpdateCounts() برای افزایش آمار مختلف هر صفحه و ب دانلود شده، فراخوانی می‌شد:

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"] [hd.exit_state()]++;           // e.g. "SUCCESS" or "FAILURE"
    counts["Http Response"] [hd.http_response()]++;     // e.g. "404 NOT FOUND"
    counts["Content-Type"] [hd.content_type()]++;        // e.g. "text/html"
}
```

خب، این همان چیزی است که می‌خواستیم، کد به نظر برسد!

در واقع، شئ HttpDownload هیچ یک از متدهای نشان داده شده را نداشت. در عوض، HttpDownload یک کلاس خیلی بزرگ و پیچیده، با تعداد زیادی کلاس‌های تودرتو بود و مجبور بودیم خودمان این مقدارها را بیرون بکشیم. گاهی اوقات این مقدارها کاملاً از بین رفته و اوضاع خیمتر می‌شد، در این حالت از عبارت unknown به عنوان مقدار پیش‌فرض استفاده می‌کردیم.

به این دلایل، کد واقعی کاملاً آشفته بود:

```
// WARNING: DO NOT STARE DIRECTLY AT THIS CODE FOR EXTENDED PERIODS OF TIME.

void UpdateCounts(HttpDownload hd) {
    // Figure out the Exit State, if available.
    if (!hd.has_event_log() || !hd.event_log().has_exit_state()) {
        counts["Exit State"] ["unknown"]++;
    } else {
        string state_str = ExitStateTypeName(hd.event_log().exit_state());
        counts["Exit State"] [state_str]++;
    }
    // If there are no HTTP headers at all, use "unknown" for the remaining elements.
    if (!hd.has_http_headers()) {
        counts["Http Response"] ["unknown"]++;
        counts["Content-Type"] ["unknown"]++;
        return;
    }
    HttpHeaders headers = hd.http_headers();
    // Log the HTTP response, if known, otherwise Log "unknown"
```

^۱ web-crawling

```

if (!headers.has_response_code()) {
    counts["Http Response"]["unknown"]++;
} else {
    string code = StringPrintf("%d", headers.response_code());
    counts["Http Response"][code]++;
}
// Log the Content-Type if known, otherwise log "unknown"
if (!headers.has_content_type()) {
    counts["Content-Type"]["unknown"]++;
} else {
    string content_type = ContentTypeMime(headers.content_type());
    counts["Content-Type"][content_type]++;
}
}

```

همان گونه که مشاهده می‌کنید، در اینجا خطوط کد بسیار زیاد و تعداد زیادی منطق و حتی چند خط تکراری از کد وجود دارد. این کد برای خواندن جالب نیست.

اشکال اساسی این کد این است که بین وظیفه‌های مختلف، عقب و جلو می‌رود. در اینجا وظایف مختلفی وجود دارد که در کل کد در هم تنیده شده‌اند:

۱. استفاده از `unknown` به عنوان مقدار پیش‌فرض برای هر `key`
۲. تشخیص اینکه اعضای `HttpDownload` از دست رفته است یا نه.
۳. استخراج مقدار و تبدیل آن به یک رشته.
۴. به روزرسانی `.counts[]`

می‌توانیم با جداسازی برخی از این وظیفه‌ها به مناطق مجزا، کد را بهبود دهیم:

```

void UpdateCounts(HttpDownload hd) {
    // Task: define default values for each of the values we want to extract
    string exit_state = "unknown";
    string http_response = "unknown";
    string content_type = "unknown";
    // Task: try to extract each value from HttpDownload, one by one
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        exit_state = ExitStateTypeName(hd.event_log().exit_state());
    }
}

```

```

if (hd.has_http_headers() && hd.http_headers().has_response_code()) {
    http_response = StringPrintf("%d", hd.http_headers().response_code());
}
if (hd.has_http_headers() && hd.http_headers().has_content_type()) {
    content_type = ContentTypeMime(hd.http_headers().content_type());
}
// Task: update counts[]
counts["Exit State"][exit_state]++;
counts["Http Response"][http_response]++;
counts["Content-Type"][content_type]++;
}

```

همان گونه که مشاهده می‌کنید، کد دارای سه منطقه جداگانه با اهداف زیر است:

۱. تعریف پیشفرضها برای سه کلید مورد نظر ما.
۲. استخراج مقدارها (در صورت موجود بودن) برای هر یک از این کلیدها و تبدیل آنها به رشته.
۳. به روزرسانی [count برای هر کلید/مقدار (key/value)

مزیت این مناطق این است که آنها از یکدیگر جداسازی شده‌اند و زمانی که در حال خواندن یک منطقه هستید، لازم نیست در مورد دیگر مناطق فکر کنید.

توجه داشته باشید که اگرچه ما چهار وظیفه را لیست کرده بودیم ولی تنها قادر به جداسازی سه مورد از آنها شدیم. این کاملاً خوب است، در واقع وظیفه‌هایی که در ابتدا لیست می‌کنید، فقط یک نقطه شروع بوده و حتی جدا کردن برخی از آنها (و نه همه آنها) می‌تواند کمک زیادی کند.

بهبودهای بیشتر^۱

این نسخه جدید کد بهبود قابل توجهی نسبت به هیولای اولیه دارد و ما حتی در انجام این پاکسازی نیازی به ایجاد توابع دیگری نداشتیم. همان گونه که قبلاً اشاره کردیم، ایده «یک وظیفه در هر لحظه» می‌تواند به شما برای پاکسازی کد، صرف نظر از مرزهای توابع کمک کند.

با این حال، ما همچنان می‌توانستیم این کد را با روش دیگری، یعنی با معرفی سه تابع کمکی بهبود بخشیم:

^۱ Further Improvements

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"][[ExitState(hd)]]++;
    counts["Http Response"][[HttpResponse(hd)]]++;
    counts["Content-Type"][[ContentType(hd)]]++;
}
```

این توابع مقدار متناظر را استخراج کرده و یا مقدار `unknown` را بر می‌گردانند. به عنوان مثال:

```
string ExitState(HttpDownload hd) {
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        return ExitStateTypeName(hd.event_log().exit_state());
    } else {
        return "unknown";
    }
}
```

توجه داشته باشید که این راه حل جایگزین، حتی هیچ متغیری را تعریف نمی‌کند! همان گونه که در فصل نهم، قسمت متغیرها و خوانایی اشاره کردیم، می‌توان متغیرهایی که نتایج واسط (یا موقت) را نگهداری می‌کنند، به طور کامل حذف کرد.

در این راه حل، به سادگی مشکل را در جهت دیگری تکه کرده‌ایم. هر دو راه حل خوانایی قابل توجهی دارند و خواننده هنگام خواندن کد تنها به یک وظیفه در یک زمان فکر می‌کند.

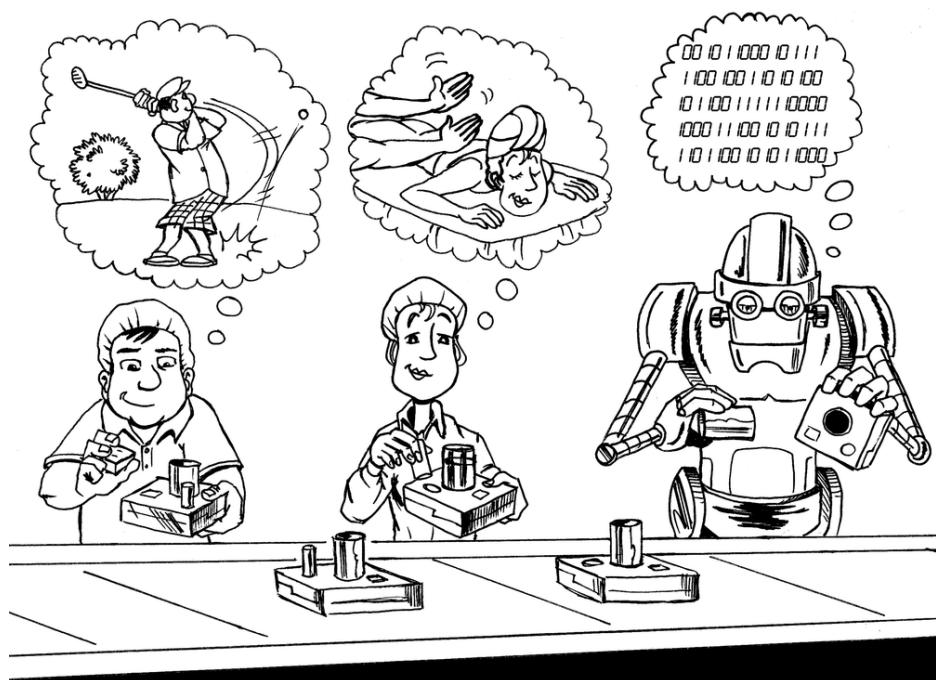
خلاصه فصل

در این فصل یک تکنیک ساده برای سازماندهی کد یعنی «تنها یک وظیفه را در یک زمان انجام دهیم» ارائه گردید.

شاید برخی از این وظیفه‌ها به سادگی به توابع یا کلاس‌های جداگانه تبدیل شوند. بقیه ممکن است فقط یک پاراگراف منطقی در یک تابع تکی باشند. جزئیات دقیق چگونگی جداسازی این وظیفه‌ها به اندازه واقعیت جدا شدن آن‌ها مهم نیست بلکه قسمت سخت این کار توصیف دقیق همه کارهای کوچکی است که برنامه شما انجام می‌دهد.

فصل دوازدهم

تبديل افکار به کد



زمانی شما واقعاً چیزی را فهمیده‌اید که بتوانید آن را به مادربرگ خود توضیح دهید.
(آلرت اینیشتین)

هنگام توضیح یک ایده بیچیده برای یک شخص، بیان همه جزئیات به راحتی او را دچار سردرگمی می‌کند. تسلط بر توضیح یک ایده به زبان ساده به گونه‌ای که شخصی که از شما دانش کمتری دارد، بتواند آن را درک کند، مهارتی ارزشمند است. لازمه این کار، توان خلاصه کردن یک ایده در مهم‌ترین مفاهیم آن است. انجام این کار علاوه بر کمک به درک دیگران برای شما نیز این فایده را دارد که می‌توانید در مورد ایده خود با وضوح بیشتری فکر کنید.

در هنگام ارائه کد به خواننده باید از این مهارت استفاده کنید. معتقد‌یم مهم‌ترین وسیله برای توضیح نحوه کارکرد برنامه شما، سورس کدتان است، بنابراین باید کد را به زبان انگلیسی ساده بنویسید.

در این فصل از فرآیندی ساده استفاده خواهیم کرد که می‌تواند به شما در نوشتن یک کد واضح‌تر کمک کند:

۱. به عنوان یک همکار آنچه را کد برای انجام نیاز دارد به زبان ساده، توضیح دهید.
۲. به کلمات کلیدی و عبارت‌های استفاده شده در این توضیحات توجه کنید.
۳. کد خود را مطابق با این توضیحات بنویسید.

توضیح منطق کد به طور شفاف

در اینجا قطعه‌ای از یک کد به زبان PHP از یک صفحه وب داریم. این کد در بالای یک صفحه امن قرار دارد که بررسی می‌کند آیا کاربر مجاز^۱ به دیدن این صفحه است یا نه؟ اگر مجاز نباشد، بلافاصله صفحه‌ای را نشان می‌دهد که به کاربر می‌گوید اجازه دیدن صفحه را ندارد:

```
$is_admin = is_admin_request();
if ($document) {
    if (!$is_admin && ($document['username'] != $_SESSION['username']))
        return not_authorized();
}
else {
    if (!$is_admin) {
```

^۱ Authorize

```

        return not_authorized();
    }
}

// continue rendering the page ...

```

منطق این کد هنوز به طور کامل، تکمیل نشده است. همان گونه که از بخش دوم(ساده سازی حلقه‌ها و منطق) به یاد دارید، درک چنین منطق‌های تودرتویی ساده نیست. احتمالاً منطق این کد می‌تواند ساده‌تر شود، اما چگونه؟ اجازه دهید با توصیف منطق به زبان ساده شروع کنیم:

دو روش برای مجاز بودن شما جهت دیدن صفحه وب وجود دارد:

۱. شما admin هستید.

۲. شما مالک سند فعلی هستید(اگر سندی وجود داشته باشد)

در غیر این صورت، شما مجاز نیستید.

روشی جایگزین با الهام از این توضیحات این است:

```

if (is_admin_request()) {
    // authorized
} elseif ($document && ($document['username'] == $_SESSION['username'])) {
    // authorized
} else {
    return not_authorized();
}

// continue rendering the page ...

```

هر چند این نسخه به دلیل وجود دو قسمت خالی کمی غیرمعمول است، اما دارای کدی کوتاه‌تر و منطقی ساده‌تر است، زیرا هیچ نفی^۱ کردنی وجود ندارد(در حالی که راه حل قبلی دارای سه «not» بود). همچنین درک آن نیز آسان‌تر می‌باشد.

^۱ negation

اطلاع از کتابخانه‌های تان می‌تواند مفید باشد

زمانی ما یک وبسایت داشتیم که شامل یک «جعبه نکات^۱» بود و پیشنهادات مفیدی را به کاربر نشان می‌داد. مانند:

Tip: Log in to see your past queries. [Show me another tip!]

در آن جا هزاران نکته وجود داشت که همگی در داخل کد HTML مخفی شده بود:

```
<div id="tip-1" class="tip">Tip: Log in to see your past queries.</div>
<div id="tip-2" class="tip">Tip: Click on a picture to see it close up.</div>
...
...
```

هنگامی بازدید یک کاربر از صفحه، یکی از این div‌ها به صورت تصادفی قابل نمایش(visible) شده و بقیه div‌ها در حالت مخفی(hidden) باقی می‌مانند و اگر روی لینک «Show me another tip!» کلیک شود، نکته بعدی نمایش داده خواهد داد. در اینجا برخی از کدهایی که برای پیاده سازی این ویژگی با استفاده از کتابخانه jQuery نوشته شده بود، آورده شده است:

```
var show_next_tip = function () {
    var num_tips = $('.tip').size();
    var shown_tip = $('.tip:visible');
    var shown_tip_num = Number(shown_tip.attr('id').slice(4));
    if (shown_tip_num === num_tips) {
        $('#tip-1').show();
    } else {
        $('#tip-' + (shown_tip_num + 1)).show();
    }
    shown_tip.hide();
};
```

هرچند این کد خوبی است اما می‌تواند بهتر از این شود. اجازه دهید ابتدا در جملاتی توصیف کیم که این کد(هنگام کلیک کاربر) تلاش می‌کند چه کاری انجام دهد:

نکته فعلی قابل مشاهده را پیدا کرده و آن را مخفی نکرد.
سپس نکته بعدی را پیدا کرده و آن را نمایش دهید.
اگر نکته بعدی وجود نداشت به ابتدای چرخه بر می‌گردیم.

^۱ tips box

راه حل جدید، مبتنی بر این توضیحات به این صورت است:

```
var show_next_tip = function () {
    var cur_tip = $('.tip:visible').hide(); // find the currently visible tip and
    // hide it
    var next_tip = cur_tip.next('.tip'); // find the next tip after it
    if (next_tip.size() === 0) { // if we've run out of tips,
        next_tip = $('.tip:first'); //      cycle back to the first tip
    }
    next_tip.show(); // show the new tip
};
```

این راه حل علاوه بر داشتن خطوط کمتر، نیازی به دستکاری مستقیم مقادیر integer ندارد. همچنین تطابق بیشتری با نحوه تفکر یک شخص در مورد آن دارد. در این راهکار استفاده از متodi next()، که متعلقjQuery است، کمک کننده است. به یاد داشته باشید که یکی از ابزارهای مختصر نویسی کد این است که، از آنچه که کتابخانه شما عرضه کرده است، آگاه باشید.

اعمال این رویکرد در مسئله‌های بزرگ‌تر

مثال‌های قبلی فرآیند ما را در بلوک‌های کوچکی از کد اعمال کرده اند. در مثال بعدی، ما آن‌ها را در یک تابع بزرگ‌تر اعمال می‌کنیم. همان‌گونه که خواهید دید، این متد می‌تواند با شناسایی بخشی از کد، که می‌تواند شکسته شود، به شما کمک کند. تصور کنید که سیستمی برای ثبت خرید سهام داریم. هر تراکنش چهار قسمت از داده را دارد:

- time (یک تاریخ دقیق و زمان خرید)
- ticker_symbol (e.g., GOOG)
- price (e.g., \$600)
- number_of_shares (e.g., 100)

به دلایل عجیبی، داده‌ها در سه جدول جداگانه دیتابیس، بصورت زیر پخش شده و در هر یک از آن‌ها، کلید اصلی^۱ منحصر به فرد^۲ است.

^۱ primary key

^۲ unique

time	ticker_symbol
3:45	IBM
3:59	IBM
4:30	GOOG
5:20	AAPL
6:00	MSFT

time	price
3:45	\$120
4:30	\$600
5:00	\$25
5:20	\$200
6:00	\$25

time	number_of_shares
3:45	50
3:59	200
4:10	75
4:30	100
5:20	80

حال ما باید برنامه‌ای برای پیوستن^۱ سه جدول به یکدیگر بنویسیم (همان گونه که یک عملیات JOIN در SQL انجام می‌شود). این مرحله باید ساده باشد زیرا سطرها همه بر اساس time مرتب شده‌اند اما متسافانه برخی از سطرها مفقود شده‌اند. شما می‌خواهید همه سطرهایی که در آن‌ها با یکدیگر مطابقت دارد را پیدا کنید، و هر سطری را که نمی‌توان مرتب کرد را نادیده^۲ بگیرید.

در اینجا کدی به زبان Python داریم که همه سطرهای منطبق را پیدا می‌کند:

```
def PrintStockTransactions():
    stock_iter = db_read("SELECT time, ticker_symbol FROM ...")
    price_iter = ...
    num_shares_iter = ...
    # Iterate through all the rows of the 3 tables in parallel.
    while stock_iter and price_iter and num_shares_iter:
        stock_time = stock_iter.time
        price_time = price_iter.time
        num_shares_time = num_shares_iter.time
        # If all 3 rows don't have the same time, skip over the oldest row
        # Note: the "<=" below can't just be "<" in case there are 2 tied-oldest.
        if stock_time != price_time or stock_time != num_shares_time:
            if stock_time <= price_time and stock_time <= num_shares_time:
                stock_iter.NextRow()
            elif price_time <= stock_time and price_time <= num_shares_time:
                price_iter.NextRow()
            elif num_shares_time <= stock_time and num_shares_time <= price_time:
                num_shares_iter.NextRow()
            else:
                assert False # impossible
```

^۱join

^۲ignore

```

continue

assert stock_time == price_time == num_shares_time

# Print the aligned rows.
print "@", stock_time,
print stock_iter.ticker_symbol,
print price_iter.price,
print num_shares_iter.number_of_shares
stock_iter.NextRow()
price_iter.NextRow()
num_shares_iter.NextRow()

```

هر چند این کد کار می‌کند، اما چیزهای زیادی در مورد چگونگی گذر حلقه از سطرهای غیرمنطبق^۱ وجود دارد. ممکن است برخی از علائم هشدار در ذهن شما خاموش شده باشد: آیا این کد می‌تواند برخی از سطرهای را از دست بدهد؟ آیا ممکن است که مقدار قبلی، پایان یک جریان را در هر تکرار بخواند؟ چگونه می‌توانیم این کد را خواناتر کنیم؟

یک توضیح به زبان ساده از راه حل

بار دیگر، بباید به عقب بازگشته و به زبان ساده توضیح دهیم که در صدد انجام چه کاری هستیم: ما در حال خواندن سه سطر تکرار شونده به شکل موازی هستیم.

هر زمان که time سطرهای منطبق نیستند، سطرهای را پیش ببرید تا آنها منطبق شوند.

سپس سطرهای هم‌تراز شده را چاپ کرده و دوباره سطرهای را پیش ببرید.

این کار را تا زمانی که هیچ سطر منطبقی باقی نماند، ادامه دهید.

به عقب برگردید و به کد اصلی نگاه کنید. آشفته‌ترین قسمت، بلوک تعامل با «قسمت پیش‌روی سطرهای تا آن‌ها با هم منطبق شوند» بود. برای نمایش کد با شفافیت بیشتر، می‌توانیم منطق آشفته را از کد استخراج کرده و در تابع جدیدی به نام AdvanceToMatchingTime() قرار دهیم.

در اینجا نسخه جدیدی از کد وجود دارد که از این تابع جدید استفاده می‌کند:

```

def PrintStockTransactions():
    stock_iter = ...
    price_iter = ...

```

^۱ unmatched

```

num_shares_iter = ...
while True:
    time = AdvanceToMatchingTime(stock_iter, price_iter, num_shares_iter)
    if time is None:
        return
    # Print the aligned rows.
    print "@", time,
    print stock_iter.ticker_symbol,
    print price_iter.price,
    print num_shares_iter.number_of_shares
    stock_iter.NextRow()
    price_iter.NextRow()
    num_shares_iter.NextRow()

```

همان گونه که مشاهده می‌کنید، درک این کد بسیار ساده‌تر است، زیرا ما همه جزئیات مبهم، در مورد تطابق سطرها را مخفی کردیم.

اعمال متده به صورت بازگشتنی^۱

تصور اینکه چگونه AdvanceToMatchingTime() را می‌نویسید ساده است. در بدترین حالت، بسیار شبیه به بلوک کد زشته است که در نسخه اول موجود است:

```

def AdvanceToMatchingTime(stock_iter, price_iter, num_shares_iter):
    # Iterate through all the rows of the 3 tables in parallel.
    while stock_iter and price_iter and num_shares_iter:
        stock_time = stock_iter.time
        price_time = price_iter.time
        num_shares_time = num_shares_iter.time
        # If all 3 rows don't have the same time, skip over the oldest row
        if stock_time != price_time or stock_time != num_shares_time:
            if stock_time <= price_time and stock_time <= num_shares_time:
                stock_iter.NextRow()
            elif price_time <= stock_time and price_time <= num_shares_time:
                price_iter.NextRow()
            elif num_shares_time <= stock_time and num_shares_time <= price_time:
                num_shares_iter.NextRow()
            else:

```

^۱ Applying the Method Recursively

```

    assert False # impossible
    continue
    assert stock_time == price_time == num_shares_time
    return stock_time

```

حال بباید این کد را با اعمال شیوه خودمان در `AdvanceToMatchingTime()` بهبود دهیم. در اینجا توضیحی در مورد آنچه که این متدهای انجام دهد را داریم:

به زمان هر سطر فعلی نگاه کنید: اگر آن‌ها تراز^۱ شده‌اند، کار ما تمام است.
در غیر این صورت، به سمت سطرهای قبلی بروید.
این کار را تا زمانی که همه سطرهای تراز شده باشند (یا یکی از این تکرارها به پایان برسد) ادامه دهید.

این توضیحات بسیار واضح، زیبا و موزون‌تر از کد قبلی است. نکته قابل توجه این است که این توضیحات هرگز اشاره‌ای به `stock_iter` یا دیگر جزئیات خاص در مورد مشکل ما نمی‌کند. این بدان معنی است که ما می‌توانیم متغیرها را نیز به شکل ساده‌تر و عمومی‌تری تغییر نام دهیم. در اینجا نتیجه این کد را می‌بینیم:

```

def AdvanceToMatchingTime(row_iter1, row_iter2, row_iter3):
    while row_iter1 and row_iter2 and row_iter3:
        t1 = row_iter1.time
        t2 = row_iter2.time
        t3 = row_iter3.time
        if t1 == t2 == t3:
            return t1
        tmax = max(t1, t2, t3)
        # If any row is "behind," advance it.
        # Eventually, this while loop will align them all.
        if t1 < tmax: row_iter1.NextRow()
        if t2 < tmax: row_iter2.NextRow()
        if t3 < tmax: row_iter3.NextRow()
    return None # no alignment could be found

```

^۱ aligned

این کد خیلی شفافتر از کد قبلی است. الگوریتم آن ساده‌تر شده و شرط‌های پیچیده کمتری دارد. ما از نام‌های کوتاهی مانند `t1` استفاده کردیم که دیگر نیازی به فکر کردن در مورد ستون‌های^۱ خاص دیتابیس ندارد.

^۱ columns

خلاصه فصل

در این فصل در مورد تکنیک ساده توصیف برنامه به زبان ساده و استفاده از این توصیف‌ها برای کمک به نوشتن کد ساده‌تر بحث کردیم. این تکنیک به شکل فریبنده‌ای ساده، اما بسیار مفید است. نگاه کردن به کلمات و عبارت‌های استفاده شده در توصیفات می‌تواند در تشخیص اینکه کدام زیرمسئله‌ها باید شکسته شوند، به شما کمک کند.

فرآیند «بیان کردن موارد به زبان ساده» کاربردهای دیگری نیز غیر از کمک در نوشتن کد دارد، به عنوان مثال یکی از همکاران ما در آزمایشگاه کامپیوتر بیان می‌کرد که، هر گاه یک دانشجو برای اشکال‌زدایی برنامه‌اش به کمک نیاز داشت، ابتدا باید مشکل را به یک خرس عروسکی که در گوشه اتاق بود، توضیح می‌داد. با کمال تعجب در اکثر موارد توصیف مسئله با صدای بلند، به دانشجو کمک می‌کرد تا متوجه راه حل شود. این تکنیک به اسم اردک پلاستیکی^۱ معروف است.

به یاد داشته باشید که اگر نمی‌توانید مشکل یا طراحی خود را با کلمات توصیف کنید، احتمالاً چیزی را در نظر نگرفته‌اید یا در برنامه تعریف نشده است. بیان یک برنامه یا یک ایده با کلمات، می‌تواند آن را مجبور کند که به خودش شکل بگیرد.

^۱ rubber ducking

فصل سیزدهم

نوشتن کد کمتر



دانستن اینکه چه زمانی نباید کد بنویسید، شاید مهمترین مهارتی باشد که یک برنامه‌نویس می‌تواند یاد بگیرد. هر خط از کدی که می‌نویسید خطی است که باید مورد تست قرار گرفته و نگه‌داری شود. با استفاده مجدد از کتابخانه‌ها یا حذف ویژگی‌ها، می‌توانید در وقت خود صرفه جویی کرده و کدپایه خود را کوتاه و با معنی نگه دارید.

کلید طلایی
کدی با حداقل خوانایی، اصلاً کد نیست.

برای پیاده سازی برخی ویژگی‌ها خود را به زحمت نیندازید، به آن نیازی ندارید

هنگامی که یک پروژه را شروع می‌کنید، طبیعی است که هیجان زده شده و به تمام ویژگی‌های هیجان‌انگیز که می‌خواهید پیاده سازی کنید فکر کنید. برنامه‌نویسان تمایل دارند که تعداد ویژگی‌هایی که واقعاً برای پروژه ضروری است را دست بالا بگیرند اما در پایان بسیاری از این ویژگی‌ها ناتمام یا بلا استفاده می‌مانند و یا اینکه تنها برنامه را پیچیده می‌کنند.

از سوی دیگر برنامه‌نویسان مایل‌اند مقدار تلاشی که برای پیاده سازی یک ویژگی نیاز است را دست کم بگیرند. آن‌ها در خوش‌بینانه‌ترین حالت می‌توانند تخمین بزنند که چه مدت طول می‌کشد تا یک نمونه اولیه^۱ خام را پیاده سازی کنند، اما بی‌شک مدت زمان اضافه‌ای که درگیر نگه‌داری^۲ برنامه در آینده و مستندسازی^۳ و افزایش حجم کدپایه را فراموش می‌کنند.

شکستن نیازمندی‌ها با مطرح کردن سوالات مختلف

لازم نیست همه برنامه‌نویسان به طور کامل و صحیح، قادر به مدیریت همه ورودی‌ها باشند. اگر شما نیازمندی‌هایتان را واقعاً موشکافی^۴ کنید، گاهی می‌توانید یک مشکل ساده را که به کد کمتری نیاز دارد از برنامه جدا کرده و بطور مستقل آن را بررسی نمایید. بباید چند مثال از این مورد را بررسی کنیم.

مثال: فروشگاه یاب

^۱ prototype

^۲ maintenance

^۳ documentation

^۴ scrutinize

فرض کنید که در حال نوشتن یک برنامه فروشگاه یاب برای یک کسب و کار هستید. در ابتدا شما فکر می‌کنید که برای پیاده سازی این برنامه باید هر طول^۱ و عرض^۲ جغرافیایی داده شده کاربر، نزدیک‌ترین فروشگاه به آن را پیدا کرد. بنابراین برای انجام چنین کاری به صحیح‌ترین شکل ممکن، باید موارد زیر را مدیریت کنید:

- وقتی که مکان‌ها^۳ در هر دو طرف خط بین‌المللی زمان^۴ قرار دارند.
- وقتی که مکان‌ها نزدیک قطب شمال یا قطب جنوب قرار دارند.
- برای تنظیم انحنا یا خمیدگی کره زمین، درجه‌های طولی به ازاء هر مایل، تغییر می‌کنند.

مدیریت همه این موارد، نیازمند مقداری دقت و چشم پوشی از برخی نقاط مرزی دارد. حال اگر برای اپلیکیشن شما، تنها^۵ فروشگاه در ایالت Texas وجود داشته باشد. سه مشکل بیان شده در لیست بالا برای این منطقه کوچک، مهم نیستند. بنابراین می‌توانید نیازمندی‌های خود را به صورت زیر کاهش دهید:

- برای یک کاربر نزدیک Texas، نزدیک‌ترین فروشگاه در Texas را پیدا کن.

حل این مسئله ساده‌تر است، زیرا می‌توان فقط با تکرار روی هر فروشگاه و محاسبه فاصله اقلیدسی^۶ بین طول و عرض جغرافیایی، راه حل را پیدا کرد.

مثال: افزودن حافظه نهان یا Cache

زمانی یک برنامه Java داشتیم که به شکل متناوب شئ‌ها را از روی دیسک می‌خواند و سرعت برنامه محدود به سرعت خواندن دیسک شده بود، بنابراین ما می‌خواستیم یک نوع مرتب‌سازی سیستم حافظه موقت(Cache) را پیاده سازی کنیم. یک دنباله معمولی از خواندن‌ها شبیه زیر بود:

```
read Object A
read Object A
read Object A
read Object B
read Object B
read Object C
```

^۱ longitude

^۲ latitude

^۳ locations

^۴ International Date Line(IDL)

^۵ Euclidean

```
read Object D
read Object D
```

همان گونه که می‌توانید ببینید، دسترسی‌های مکرر به یک شیء مشابه انجام شده است، بنابراین کردن اطلاعات، قطعاً به ما کمک می‌کرد.

زمانی که با این مشکل روبرو شدیم، اولین حس غریزی ما این بود که از یک cache استفاده کنیم که آیتم‌های کمتر استفاده شده‌ای اخیر را رها^۱ کند. هر چند در کتابخانه خود چنین سیستمی را نداشتیم و مجبور بودیم خودمان آن را پیاده سازی کنیم. ولی با مشکلی چندانی روبرو نبودیم، چرا که قبل از این ساختاردادهای^۲ را که شامل دو جدول هش^۳ و یک لیست پیوندی^۴ تکی بود (و شاید در کل ۱۰۰ خط کد بود) را پیاده سازی کرده بودیم.

در عین حال متوجه شدیم که دسترسی‌های مکرر همیشه به یک سطر انجام می‌شود، بنابراین به جای پیاده سازی یک cache فقط یک LRU cache تک آیتمی را پیاده سازی کردیم:

```
DiskObject lastUsed; // class member
DiskObject lookUp(String key) {
    if (lastUsed == null || !lastUsed.key().equals(key)) {
        lastUsed = loadDiskObject(key);
    }
    return lastUsed;
}
```

بدون اینکه کدنویسی زیادی را انجام دهیم، این پیاده سازی سبب بهبود ۹۰ درصدی بود، و برنامه نیز از مقدار حافظه کمی استفاده می‌کرد.

فواید «حذف نیازمندی‌ها» و «حل مسئله‌های ساده‌تر» نباید اغراق‌آمیز باشد. نیازمندی‌ها اغلب با شیوه‌های ظریفی با یکدیگر تداخل دارند. این یعنی، مدت زمان حل نیمی از مسئله، نسبت کدنویسی، حدوداً یک چهارم است.

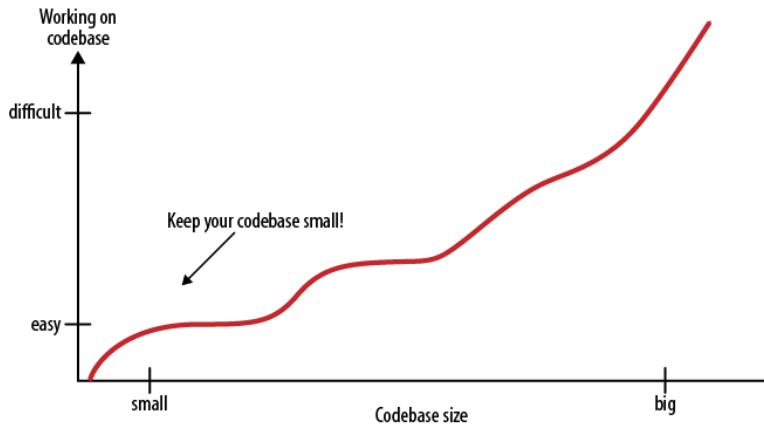
^۱ discards

^۲ data structure

^۳ hash table

^۴ linked list

کدپایه خود را کوچک نگه دارید



وقتی که برای اولین بار یک پروژه نرم افزاری را شروع کرده و تنها یک یا دو فایل منبع^۱ دارید، همه چیز عالی است. کامپایل و اجرای کد خیلی سریع است، تغییرات به سادگی انجام می‌شود و به خاطر سپردن اینکه هر تابع یا کلاس کجا تعریف شده است راحت است.

با رشد پروژه، رفته رفته دایرکتوری پروژه شما با فایل‌های منبع بیشتری پر می‌شود. به زودی به چندین دایرکتوری برای سازماندهی همه آن‌ها نیاز پیدا خواهد کرد. یادآوری اینکه کدام توابع چه توابع دیگری را فراخوانی می‌کنند سخت‌تر شده و همچنین ردیابی و برطرف کردن اشکالات، کار بیشتری را می‌طلبد.

در نهایت، شما کد منبع بزرگی دارید که در تعداد بسیاری از دایرکتوری‌های مختلف پخش شده‌اند. این پروژه خیلی بزرگ است و هیچ کس به تنهایی همه آن را نمی‌فهمد. افزودن ویژگی‌های جدید در دنک می‌شود و کار با کد، دست و پا گیر و ناخوشایند خواهد بود.

آنچه که توصیف کردیم یک قانون طبیعی در جهان است. با رشد خصوصیات یک سیستم، پیچیدگی مورد نیاز، جهت نگه‌داری خصوصیات در کنار هم، خیلی سریع‌تر رشد می‌کند. بهترین راه برای اینکه از عهده این کار برآید این است که کدپایه خود را حتی با رشد پروژه، تا حد ممکن کوچک و سبک^۲ نگه دارید. بنابراین:

^۱ source files

^۲ lightweight

- تا جای امکان باید کد «کاربردی^۱» بیشتری ایجاد کنید تا بتوانید کدهای تکراری را حذف کنید(فصل ۱۰ را مشاهده کنید).
- کدهای استفاده نشده یا ویژگی‌های بی‌فایده را حذف کنید(در ادامه توضیح می‌دهیم).
- پروژه خود را به صورت جدا در دسته‌بندی‌های مختلف، در زیر پروژه‌های جداگانه نگه دارید.
- به طور کلی، از وزن (حجم) کدپایه خود آگاه باشید و آن را سبک و چابک^۲ نگه دارید.

حذف کدهای بی‌فایده

bagban‌ها اغلب برای زنده نگه‌داشتن و رشد گیاهان، آن‌ها را هرس می‌کنند. در برنامه‌نویسی هم این ایده خوبی است که کدهای بی‌فایده را هرس کنیم.

کدنویسان اغلب تمایلی برای حذف کدهای نوشته شده ندارند، حذف بخشی از کد، به این معنی است که قبول کنیم مدت زمانی که صرف آن شده بود، اتلاف وقت بوده است. ولی با این حال، این کار را انجام دهید! عکاسان، نویسنده‌ها و کارگردانان نیز همه کارهای خود را نگه نمی‌دارند.

حذف توابع مجزا آسان است، اما گاهی «کد استفاده نشده» در کل پروژه تنیده شده و برای شما مجھول است. در اینجا چند مثال آورده‌ایم:

- شما سیستم اصلی خود را برای مدیریت نام‌های بین‌المللی برای فایل‌ها طراحی کرده‌اید و اکنون کد توسط تبدیلات، کدهای جدیدتری تولید کرده است. با این حال، این کد کاملاً کاربردی نیست و برنامه شما هیچ گاه با نام‌های بین‌المللی، مورد استفاده قرار نگرفته است. چرا چنین قابلیتی را حذف نکنیم؟
- شما می‌خواهید حتی اگر سیستم با کمبود حافظه^۳ مواجه شد، برنامه کار کند، بنابراین منطق‌های هوشمند زیادی نوشته‌اید که سعی می‌کند برنامه را در شرایط کمبود حافظه بازیابی کند. این ایده خوب است اما در عمل وقتی برنامه در شرایط کمبود حافظه اجرا شود، برنامه شما به هر حال به یک زامبی ناپایدار^۴ تبدیل می‌شود. در این حالت همه ویژگی‌های هسته غیر قابل استفاده شده و فاصله برنامه تا مرگ آن تنها یک کلیک موس خواهد بود. پس چرا برنامه را تنها با یک پیام ساده «متاسفیم! سیستم دچار کمبود حافظه شده است» خاتمه ندهیم و همه کد مربوط به کمبود حافظه را حذف نکنیم؟

^۱ utility

^۲ nimble

^۳ Out of memory

^۴ unstable zombie

با کتابخانه‌های موجود در اطراف خود آشنا باشید

خیلی از وقت‌ها، برنامه‌نویسان اطلاع ندارند که با کتابخانه‌های موجود می‌توانند مشکل خود را حل کنند. همچنین گاهی فراموش می‌کنند که یک کتابخانه چه کاری می‌تواند انجام دهد. دانستن قابلیت‌های کد کتابخانه برای استفاده از آن بسیار مهم است.

در اینجا یک پیشنهاد خوب داریم: **هر چند وقت یکبار، ۱۵ دقیقه از وقت خود را برای خواندن نام همه توابع، ماژول‌ها و نوع‌ها در کتابخانه استاندارد خود صرف کنید.** این شامل کتابخانه الگوی استاندارد C++, Java API، ماژول‌های داخلی Python و بقیه موارد می‌شود.

هدف از این کار به خاطر سپردن کل کتابخانه نبوده و تنها برای این است که بدانید چه چیزهایی وجود دارد، به گونه‌ای که وقتی روی کد جدیدی کار می‌کنید به این فکر کنید که «صبر کن! این به نظر آشنا میرسه، من قبل این API را دیدم و». ما معتقدیم که سریعاً فایده انجام این کار را خواهید دید و در اولین فرصت تمایل خواهید داشت از این کتابخانه‌ها استفاده کنید.

مثال: لیست‌ها و مجموعه‌ها در Python

فرض کنید که شما یک لیست در Python دارید (مثلا [2, 1, 2]) و می‌خواهید لیستی از عناصر بدون تکرار را استخراج کنید (در این مورد [2, 1]). می‌توانید این کار را با استفاده از یک دیکشنری، که دارای یک لیست از key‌هایی است که تضمین می‌کند منحصر به فرد باشند، به شکل زیر پیاده سازی کنید:

```
def unique(elements):
    temp = {}
    for element in elements:
        temp[element] = None # The value doesn't matter.
    return temp.keys()
unique_elements = unique([2,1,2])
```

از سوی دیگر می‌توانید فقط از Data Type `set` کمتر شناخته شده استفاده کنید:

```
unique_elements = set([2,1,2]) # Remove duplicates
```

این دقیقاً مثل یک لیست عادی قابل تکرار است. اگر واقعاً یک لیست object دیگر را خواستید، فقط کافی است به صورت زیر از آن استفاده کنید:

```
unique_elements = list(set([2,1,2])) # Remove duplicates
```

بدیهی است که set ابزار مناسبی برای کار در اینجا است. اما اگر از این نوع داده آگاه نبودید، احتمالاً کدی شبیه تابع unique() را خودتان تولید می‌کردید.

چرا استفاده مجدد از کتابخانه‌ها چنین موفقیت دارد؟

طبق آمار مهندسین نرم افزار به طور میانگین روزانه ده خط کد قابل حمل^۱ تولید می‌کنند. هنگامی که برنامه‌نویسان برای اولین بار این جمله را می‌شنوند، از قبول کردن آن طفره رفته و می‌گویند: «ده خط کد؟ من می‌توانم آن را در یک دقیقه بنویسم!». کلمه کلیدی در اینجا **قابل حمل** است. هر خط از کد در یک کتابخانه بالغ، نشان‌دهنده مقدار مناسبی از طراحی، اشکال‌زدایی، بازنویسی، مستندسازی، بهینه‌سازی و انجام تست است. هر خط از کد که از این فرآیندهای داروینی^۲ جان سالم به در ببرد، ارزش زیادی دارد. به همین دلیل است که استفاده مجدد از کتابخانه‌ها یک پیروزی محسوب می‌شود، هم در صرفه جویی زمان و هم نوشتن کد کمتر.

مثال: استفاده از ابزارهای Unix به جای نوشتن کد

زمانی که یک سرور وب به طور متناسب کدهای پاسخ HTTP 4xx یا 5xx را بر می‌گرداند، این نشان دهنده یک مشکل است(4xx خطایی از سمت کلاینت است و 5xx مربوط به زمانی است که سرور دچار خطا شده است). ما می‌خواهیم برنامه‌ای بنویسیم که لاغهای^۳ دسترسی به یک وب سرور را بررسی^۴ کرده و تشخیص دهد که کدام URL‌ها، علت بیشترین خطاها هستند.

لاغهای دسترسی معمولاً چیزی شبیه به این است:

```
1.2.3.4 example.com [24/Aug/2010:01:08:34] "GET /index.html HTTP/1.1" 200 ...
2.3.4.5 example.com [24/Aug/2010:01:14:27] "GET /help?topic=8 HTTP/1.1" 500 ...
3.4.5.6 example.com [24/Aug/2010:01:15:54] "GET /favicon.ico HTTP/1.1" 404 ...
...
```

به طور کلی، آن‌ها شامل خطهایی از این موارد هستند:

browser-IP	host	[date]	"GET /url-path HTTP/1.1"	HTTP-response-code	...
------------	------	--------	--------------------------	--------------------	-----

^۱ shippable

^۲ Darwinian

^۳ logs

^۴ parses

نوشتن برنامه‌ای برای پیدا کردن بیشترین url-path‌ها با کد پاسخ 4xx یا 5xx احتمالاً به سادگی ۲۰ خط کدنویسی در یک زبان شبیه C++ یا Java است.

به جای این کار، شما می‌توانید در Unix از دستور زیر در خط فرمان استفاده کنید:

```
cat access.log | awk '{ print $5 " " $7 }' | egrep "[45].\.$" \
| sort | uniq -c | sort -nr
```

که خروجی زیر را تولید می‌کند:

```
95 /favicon.ico 404
13 /help?topic=8 500
11 /login 403
...
<count> <path> <http response code>
```

نکته جالب در مورد این خط فرمان این است که ما از نوشتن هر کد واقعی یا بررسی هرچیزی داخل کنترل سورس خودداری کرده‌ایم.



خلاصه فصل

ماجراجویی، هیجان!! یک Jedi! این چیزها را آرزو نمی‌کند.(Yoda)

این فصل درباره کوتاه نوشتن کد تا حد ممکن است. هر خط جدیدی از کد نیازمند تست، مستندسازی و نگهداری است. علاوه بر آن، با داشتن کد زیاد، کدپایه سنگین‌تر و برای توسعه سخت‌تر خواهد شد. شما می‌توانید از نوشتن خطوط کد جدید از طریق روش‌های زیر جلوگیری کنید:

- از بین بردن ویژگی‌های غیر ضروری و بدون استفاده حتی اگر پیشرفته باشند.
- بازنگری در مورد نیازمندی‌ها برای حل مشکل با ساده‌ترین نسخه از کد.
- آگاهی داشتن در مورد کتابخانه‌های استاندارد با خواندن دوره‌ای کل API‌ها.

بخش چهارم

موضوعات برگزیده

در سه بخش قبلی، طیف گسترهای از تکنیک‌های آسانتر کردن درک را پوشش دادیم. در این بخش، می‌خواهیم بعضی از این تکنیک‌ها را برای دو موضوع برگزیده اعمال کنیم. ابتدا، قصد داریم در مورد «تست» بحث کرده و نحوه نوشتتن تست‌هایی که هم زمان موثر و خوانا باشند را مورد بررسی قرار دهیم.

در ادامه طراحی و پیاده سازی یک ساختار داده خاص-منظوره (یک شمارنده دقیقه/ساعت^۱) را بررسی می‌کنیم که در آن کارآیی، طراحی خوب و اثر متقابل خوانایی وجود دارد.

^۱ minute/hour counter

فصل چهاردهم

انجام تست و خوانایی



در این فصل، قصد داریم تکنیک‌های ساده‌ای را برای نوشتن تست‌های مرتب^۱ و موثر^۲ نشان دهیم. تست کردن به معنای چیزهای مختلف برای کاربران مختلف است. منظور ما از «تست» در این فصل استفاده از کدی است که تنها هدف آن بررسی رفتار یک قطعه کد دیگر است. هدف ما تمرکز بر روی جنبه خوانایی تست‌ها بوده و به چگونگی نوشتن تست قبل از نوشتن کد واقعی («توسعه آزمون محور^۳») و یا دیگر جنبه‌های فلسفی توسعه تست، توجهی نداریم.

خوانایی تست‌های خود را ساده و قابل نگهداری کنید

خوانایی کد تست به اندازه خوانایی کدهای غیر تست مهم است. در نگاه کدنویسان غالباً کدهای تست به عنوان مستندات^۴ غیر رسمی هستند که می‌گوید کد واقعی چگونه کار کرده و چگونه باید مورد استفاده قرار گیرد. بنابراین اگر تست‌ها برای خواندن ساده باشند، کاربران رفتارهای کد اصلی را بهتر می‌توانند درک کنند.

کلید طلایبی

کد تست باید قابل خواندن باشد تا دیگر کدنویسان با تغییر آن‌ها یا اضافه کردن تست‌های جدید، راحت باشند.

هنگامی که یک کد تست بزرگ و ترسناک باشد، اتفاقات زیر می‌افتد:

کدنویسان از تغییر کد واقعی می‌ترسند، چرا که نمی‌خواهند با این کد دچار آشفتگی شوند.
همچنین به روزرسانی همه تست‌ها برایشان یک کابوس خواهد شد.

کدنویسان وقتی که جدیدی اضافه می‌کنند، تست جدید اضافه نمی‌کنند. با گذشت زمان، مازول‌ها کمتر و کمتر تست می‌شوند و دیگر اطمینانی به کار کردن همه آن‌ها وجود نخواهد داشت.

از سوی دیگر، شما می‌خواهید کاربران (به ویژه خودتان!) را تشویق کنید که با تست کردن آن، راحت باشند. آن‌ها باید بتوانند تشخیص دهند که چرا یک تغییر جدید باعث شکست تست موجود می‌شود و همچنین احساس کنند که اضافه کردن تست جدید ساده است.

^۱ neat

^۲ effective

^۳ test driven development

^۴ documentation

مشکل این تست چیست؟!

زمانی در کدپایه ما، تابعی برای مرتب سازی و فیلتر^۱ کردن لیستی از نتایج جستجوی امتیازبندی^۲ شده وجود داشت که اعلان^۳ تابع آن به شکل زیر بود:

```
// Sort 'docs' by score (highest first) and remove negative-scored documents.
void SortAndFilterDocs(vector<ScoredDocument>* docs);
```

یک تست برای این تابع، در ابتدا چیزی شبیه زیر است:

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;
    SortAndFilterDocs(&docs);
    assert(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

حداقل هشت اشکال مختلف در این کد تست وجود دارد. در پایان فصل، شما قادر خواهید بود همه این اشکالات را تشخیص و سپس تصحیح کنید.

^۱ filter

^۲ scored

^۳ declaration

تصحیح این تست به شکل خواناتر

به عنوان یک اصل کلی در طراحی، شما باید جزئیات کم اهمیت را از دید کاربر پنهان کنید تا جزئیات مهمتر برجسته‌تر شوند.

کد تست مذکور به وضوح این اصل را نقض می‌کند. همه جزئیات این تست واضح بوده و مانند دقایق بی اهمیتی که صرف تنظیم `vector<ScoredDocument>` می‌شود! اکثر خطوط این کد شامل `url`، `score` و `docs[]` شده است که فقط جزئیاتی در مورد چگونگی تنظیم شیوه‌ای^۱ C++ است و نه درباره اینکه این تست در یک سطح بالا^۲ چه کاری انجام می‌دهد.

به عنوان اولین قدم در تمیز کردن این مثال، می‌توانیم یکتابع کمکی^۳ شبیه کد زیر بسازیم:

```
void MakeScoredDoc(ScoredDocument* sd, double score, string url) {
    sd->score = score;
    sd->url = url;
}
```

با استفاده از این تابع، کد تست ما کمی خلاصه‌تر می‌شود:

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    MakeScoredDoc(&docs[0], -5.0, "http://example.com");
    MakeScoredDoc(&docs[1], 1, "http://example.com");
    MakeScoredDoc(&docs[2], 4, "http://example.com");
    MakeScoredDoc(&docs[3], -99998.7, "http://example.com");
    ...
}
```

اما این به اندازه کافی خوب نبوده و هنوز هم جزئیات بی اهمیتی مشاهده می‌شود. به عنوان نمونه، پارامتر «`http://example.com`» هنگام مشاهده کد، یک چیز ناخوشایند است. این پارامتر همواره یک چیز یکسان است و دقیقا URL در آن هیچ اهمیتی ندارد. چرا که تنها برای معتبر بودن `ScoredDocument` پر شده است.

^۱ objects

^۲ high level

^۳ helper function

یک دیگر از جزئیات کم اهمیت، که مجبور به دیدن آن بودیم `&docs[0]` و `docs.resize(5)` و غیره است. باید تابع کمکی `MakeScoredDoc` را تغییر دهیم تا کارهای بیشتری برای ما انجام دهد و آن را `AddScoredDoc()` نامیم:

```
void AddScoredDoc(vector<ScoredDocument>& docs, double score) {
    ScoredDocument sd;
    sd.score = score;
    sd.url = "http://example.com";
    docs.push_back(sd);
}
```

با استفاده از این تابع، کد تست ما فشرده‌تر می‌شود:

```
void Test1() {
    vector<ScoredDocument> docs;
    AddScoredDoc(docs, -5.0);
    AddScoredDoc(docs, 1);
    AddScoredDoc(docs, 4);
    AddScoredDoc(docs, -99998.7);
    ...
}
```

این کد بهتر است، اما هنوز یک تست «بسیار خوانای» و قابل نوشتتن^۲ نشده است. اگر می‌خواهید تست دیگری با مجموعه‌ای جدید از `docs` امتیازبندی شده اضافه کنید، این کار نیازمند تعداد زیادی عملیات `copy/paste` است. حال سوال این است که چگونه می‌توانیم به بهبود آن ادامه دهیم؟

ساختن حداقل دستورات تست^۳

برای بهبود کد تست، باید از تکنیک `فصل دوازدهم`، یعنی تبدیل افکار به کد استفاده کنیم. باید کاری را که تست می‌سازیم کند انجام دهد را به زبان ساده توصیف کنیم:

ما یک لیست از سندهایی داریم که به صورت `[3, 1, 4, -99998.7, -5]` امتیازبندی شده‌اند. پس از `SortAndFilterDocs()` سندهای باقی مانده باید امتیازهای `[1, 3, 4]` را به ترتیب داشته باشد.

^۱ highly readable

^۲ writable

^۳ Minimal Test Statement

همان گونه که می‌بینید، در هیچ قسمتی از این توصیفات، اشاره‌ای به `vector<ScoredDocument>` نکردیم و آرایه امتیازات مهمترین چیز در اینجا بود. در حالت ایده‌آل، کد تست ما چیزی شبیه به این خواهد بود:

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3", "4, 3, 1");
```

ما قادریم که کل این تست را تنها در یک خط کد به طور کامل به اجرا درآوریم! ماهیت اغلب تست‌ها برای بررسی این است که این ورودی/شرايط، این خروجی/رفتار را از خود نشان دهد که در اکثر مواقع این هدف می‌تواند فقط در یک خط بیان شود. کوتاه نگه داشتن دستورات تست، علاوه بر مختصر و قابل خواندن نمودن آن، سبب راحتی اضافه کردن موارد تست بیشتر می‌شود.

پیاده سازی سفارشی «Minilanguages»

توجه داشته باشید که `CheckScoresBeforeAfter()` دو آرگومان به صورت رشته می‌گیرد که آرایه‌ای از امتیازات را توصیف می‌کند. در نسخه‌های جدیدتر C++ شما می‌توانید آرایه‌های لیترال^۱ را، مانند زیر تعریف کنید:

```
CheckScoresBeforeAfter({-5, 1, 4, -99998.7, 3}, {4, 3, 1});
```

از آنجا که نمی‌توانیم این کار را در لحظه انجام دهیم، امتیازها را درون یک رشته قرار می‌دهیم که با علامت «» از هم جدا شده‌اند. برای اینکه این روش کار کند، `CheckScoresBeforeAfter()` باید آرگومان‌های این رشته را تبدیل^۲ کند.

به طور کلی، تعریف یک زبان کوچک^۳ سفارشی می‌تواند روشی قدرتمند برای بیان اطلاعات خیلی زیاد در یک فضای کوچک باشد. همچنین می‌توان از `printf()` و کتابخانه‌های عبارات منظم^۴ نیز استفاده نمود.

در این مورد، نوشتتن بعضی از توابع کمکی برای تجزیه/تبدیل کردن یک لیست از اعداد جدا شده با کاما، نباید کار خیلی سختی باشد. در اینجا `CheckScoresBeforeAfter()` به صورت زیر است:

^۱ array literals

^۲ parse

^۳ minilanguage

^۴ regular expression

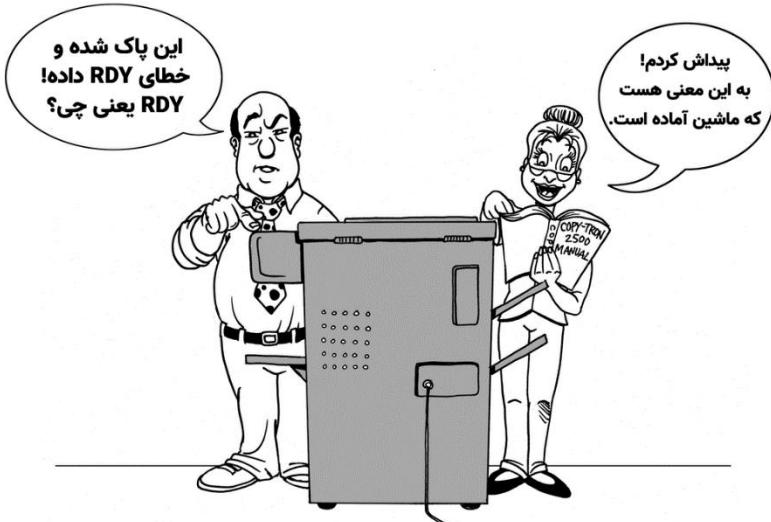
```
void CheckScoresBeforeAfter(string input, string expected_output) {
    vector<ScoredDocument> docs = ScoredDocsFromString(input);
    SortAndFilterDocs(&docs);
    string output = ScoredDocsToString(docs);
    assert(output == expected_output);
}
```

که برای کامل شدن آن، یک تابع کمکی برای تبدیل `vector<ScoredDocument>` و `string` داریم:

```
vector<ScoredDocument> ScoredDocsFromString(string scores) {
    vector<ScoredDocument> docs;
    replace(scores.begin(), scores.end(), ',', ' ');
    // Populate 'docs' from a string of space-separated scores.
    istringstream stream(scores);
    double score;
    while (stream >> score) {
        AddScoredDoc(docs, score);
    }
    return docs;
}
string ScoredDocsToString(vector<ScoredDocument> docs) {
    ostringstream stream;
    for (int i = 0; i < docs.size(); i++) {
        if (i > 0) stream << ", ";
        stream << docs[i].score;
    }
    return stream.str();
}
```

شاید در نگاه اول این کد خیلی زیاد به نظر برسد، اما کاری را که به شما اجازه می‌دهد انجام دهید، فوق العاده قدرتمند است. زیرا شما می‌توانید یک تست کلی را فقط با صدا زدن `CheckScoresBeforeAfter()` بنویسید و این امر شما را متمایل می‌کند که تست‌های بیشتری بنویسید! همان‌گونه که بعداً در این فصل این کار را انجام خواهیم داد.

پیام‌های خطرا را به شکل خوانا و صحیح بنویسید



هر چند کد قبلی خوب بود اما هنگامی که خط `assert(output == expected_output)` با شکست روبرو شود چه اتفاقی می‌افتد؟ یک پیغام خطای شبیه متن زیر تولید خواهد کرد:

```
Assertion failed: (output == expected_output),
function CheckScoresBeforeAfter, file test.cc, line 37.
```

بدیهی است که اگر شما تا کنون این خط را مشاهده نکرده باشید، نگران خواهید شد که مقدارهای `expected_output()` و `output` کجا بودند؟

استفاده از نسخه بهتری از `assert()`

خوشبختانه اکثر زبان‌ها و کتابخانه‌ها نسخه‌های سطح بالایی^۱ از `assert()` دارند که شما می‌توانید از آن‌ها استفاده کنید. بنابراین به جای نوشتن:

```
assert(output == expected_output);
```

شما می‌توانید از کتابخانه Boost در زبان C++ استفاده کنید:

```
BOOST_REQUIRE_EQUAL(output, expected_output)
```

^۱ sophisticated

حال اگر تست دچار شکست شود، پیغامی با جزئیات بیشتر دریافت خواهد کرد، که بسیار مفیدتر است:

```
test.cc(37): fatal error in "CheckScoresBeforeAfter": critical check
    output == expected_output failed ["1, 3, 4" != "4, 3, 1"]
```

در صورت در دسترس بودن، باید از متدهای assertion مفیدتر استفاده کنید چرا که در زمان شکست تست، نتایج بهتری خواهد داد.

توابع ASSERT() بهتر در زبان‌های دیگر

در Python دستور داخلی assert a == b پیغام خطای ساده‌ای را تولید می‌کند:

```
File "file.py", line X, in <module>
    assert a == b
AssertionError
```

شما می‌توانید به جای آن از متدهای unittest assertEquals() استفاده کنید:

```
import unittest
class MyTestCase(unittest.TestCase):
    def testFunction(self):
        a = 1
        b = 2
        self.assertEqual(a, b)
if __name__ == '__main__':
    unittest.main()
```

که پیام خطایی مانند عبارت زیر تولید می‌کند:

```
File "MyTestCase.py", line 7, in testFunction
    self.assertEqual(a, b)
AssertionError: 1 != 2
```

به طور خلاصه، از assert برای بررسی شرط‌هایی استفاده می‌شود که کاربر در نتایج آنها بی‌تأثیر بوده و ممکن است در زمان کامپایل/اجرا نیاز به بررسی داشته باشد (به عبارت دیگر، از assert بر روی نتیجه شرط‌هایی استفاده کنید که فقط به کدهای نوشته شده توسط برنامه‌نویس، وابسته است و تصمیمات کاربر (یا سایر موارد)، در آن بین‌تأثیر باشد). اگر شرط درون assert برقرار نباشد، برنامه در زمان اجرا متوقف شده و همین شرط در خروجی به عنوان خطای کاربر نمایش داده شود.

بسته به زبانی که در حال استفاده از آن هستید، احتمالاً یک کتابخانه یا فریمورک^۱(مانند XUnit) برای کمک به شما وجود دارد، که باید هزینه دانستن این کتابخانه‌ها را بپردازید.

پیام خطاهای دست ساز^۲

با استفاده از BOOST_REQUIRE_EQUAL()، ما قادر بودیم که پیام خطای بهتری دریافت کنیم:

```
output == expected_output failed ["1, 3, 4" != "4, 3, 1"]
```

با این حال، این پیام می‌تواند بهبود بیشتری پیدا کند.(به عنوان نمونه، می‌تواند برای دیدن ورودی اصلی که باعث این شکست شده بود، مفید باشد). پیام خطای ایده‌آل چیزی شبیه این خواهد بود:

```
CheckScoresBeforeAfter() failed,
Input:      "-5, 1, 4, -99998.7, 3"
Expected Output: "4, 3, 1"
Actual Output:   "1, 3, 4"
```

اگر این همان چیزی است که شما می‌خواهید، رو به جلو حرکت کنید و آن را بنویسید!

```
void CheckScoresBeforeAfter(...) {
    ...
    if (output != expected_output) {
        cerr << "CheckScoresBeforeAfter() failed," << endl;
        cerr << "Input:      \" " << input << "\" " << endl;
        cerr << "Expected Output: \" " << expected_output << "\" " << endl;
        cerr << "Actual Output:   \" " << output << "\" " << endl;
        abort();
    }
}
```

نکته اخلاقی این قسمت این است که پیام خطای باید تا حد امکان کمک کننده باشد. گاهی، چاپ پیام خودتان با ایجاد یک «assert» سفارشی، بهترین کار برای انجام این کار است.

انتخاب ورودی‌های خوب برای تست

^۱ framework
^۲ Hand-Crafted

انتخاب مقادیر ورودی خوب برای تست‌های شما یک هنر است. مواردی که اکنون داریم کمی اتفاقی^۱ به نظر می‌رسند:

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3", "4, 3, 1");
```

چگونه مقادیر خوبی برای ورودی انتخاب کنیم؟ ورودی‌های خوب باید کد را به طور کامل تست کرده و در ضمن باید ساده باشند تا خواندن آنها آسان باشد.

کلید طلایی

شما باید ساده‌ترین مجموعه از ورودی‌های که به طور کامل کد را به کار می‌گیرند، انتخاب کنید.

برای مثال، فرض کنید ما فقط این را نوشته بودیم:

```
CheckScoresBeforeAfter("1, 2, 3", "3, 2, 1");
```

اگرچه این تست ساده است، اما رفتار SortAndFilterDocs() را در مورد فیلتر کردن امتیازات منفی تست نمی‌کند. بنابراین اگر اشکالی در این بخش از کد وجود می‌داشت، این ورودی نمی‌توانست متوجه آن شود.

از طرف دیگر، فرض کنید ما خودمان تستی شبیه کد زیر را نوشته بودیم:

```
CheckScoresBeforeAfter("123014, -1082342, 823423, 234205, -235235",
                      "823423, 234205, 123014");
```

این مقادیر غیر مفید، پیچیده هستند و حتی تست کد را به صورت کامل انجام نمی‌دهند.

ساده سازی مقادیر ورودی

برای بهبود مقادیر ورودی چه کاری می‌توانیم انجام دهیم؟

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3", "4, 3, 1");
```

احتمالاً اولین چیزی که شما متوجه شدید این است که مقدار -99998.7- بسیار برجسته است. از آن جا که این مقدار فقط به معنای «هر عدد منفی» است، بنابراین با یک مقدار ساده‌تر همچون عدد -1- فرقی ندارد(البته اگر مقدار -99998.7- به معنی «یک عدد منفی بسیار بزرگ» بود، یک مقدار بهتر می‌توانست چیزی مانند -1e100- باشد).

¹ haphazard

کلید طلایی

مقادیر ساده و تمیز تست را که هنوز کار را به درستی انجام می‌دهند ترجیح دهید.

مقادیر دیگر در این تست خیلی بد نیستند، اما حالا که اینجا هستیم، می‌توانیم آن‌ها را تا حد امکان به ساده‌ترین عدد صحیح^۱ کاهش دهیم. همچنان، تنها یک مقدار منفی برای تست کافی است. در اینجا نسخه جدیدی از این تست را می‌توانید ببینید:

```
CheckScoresBeforeAfter("1, 2, -1, 3", "3, 2, 1");
```

ما مقادیر تست را بدون اینکه اثر آن‌ها را کمتر کنیم، ساده‌تر کردیم.

تست‌های بزرگ «شکننده»^۲

به ازای هر ورودی بزرگ و بی مفهوم یک مقدار دقیق برای تست کردن کد شما وجود دارد. به عنوان نمونه، احتمالاً شما وسوسه شده‌اید که تستی شبیه کد زیر را اضافه کنید:

```
CheckScoresBeforeAfter("100, 38, 19, -25, 4, 84, [lots of values] ...",
    "100, 99, 98, 97, 96, 95, 94, 93, ...");
```

کار خوبی که ورودی‌های بزرگ انجام می‌دهند این است که باگ‌هایی مانند سرریز بافر یا دیگر مواردی که انتظار ندارید رخ دهند را نمایش می‌دهند. اما چنین کدهایی برای مشاهده، بزرگ و ترسناک هستند و به طور کامل در انجام stress-test مفید نیستند.

^۱ integers

^۲ SMASHER

تست‌های چندگانه عملکرد

به جای ساختن یک تک ورودی «عالی» برای تست کامل کد، نوشتمن چندین تست کوچک، اغلب ساده‌تر، مفیدتر و دارای خوانایی بیشتری است. هر تست باید در یک بخش مشخص به کد شما ارسال^۱ شود و در صدد یافتن یک اشکال خاص در آن باشد. به عنوان مثال، در اینجا چهار تست برای SortAndFilterDocs وجود دارد:

```
CheckScoresBeforeAfter("2, 1, 3", "3, 2, 1");      // Basic sorting
CheckScoresBeforeAfter("0, -0.1, -10", "0");        // All values < 0 removed
CheckScoresBeforeAfter("1, -2, 1, -2", "1, 1");      // Duplicates not a problem
CheckScoresBeforeAfter("", "");                      // Empty input OK
```

اگر می‌خواهید خیلی دقیق باشید، تست‌های بیشتری نیز وجود دارد که می‌توانید بنویسید. داشتن چندین مورد تست جداگانه، کار کردن با کد را برای شخص بعدی ساده‌تر می‌کند. اگر کسی به طور اتفاقی یک باگ را معرفی کند، شکست تست، مورد خاصی که ناموفق بوده است را با دقت نشان خواهد داد.

نام‌گذاری توابع تست

کد تست به طور معمول در توابع سازماندهی شده و برای هر متده و/یا شرایطی از یک تست استفاده می‌شود. به عنوان نمونه، کد تست کننده SortAndFilterDocs() در یک تابع، به نام Test1() بود:

```
void Test1() {
    ...
}
```

شاید انتخاب یک نام خوب برای یک تابع تست خسته کننده و بی‌ربط به نظر برسد، اما به هیچ وجه به نام‌های بی معنی مانند Test1(), Test2() و موارد مشابه متอسل نشوید. در عوض، باید از نامی که جزئیات تست را شرح می‌دهد استفاده کنید. به ویژه، جزئیاتی که شخص خواننده به سرعت بتواند درک کند که:

- چه کلاسی (در صورت وجود) دارد تست می‌شود.

^۱ push

- چه تابعی دارد تست می‌شود.

- چه وضعیت یا باگی دارد تست می‌شود.

یک رویکرد ساده برای انتخاب نام خوب برای یک تابع تست، این است که تنها اطلاعات را به یکدیگر الحق^۱ کرده و در صورت امکان از یک پیشوند «Test_» استفاده کنید.

به عنوان نمونه به جای نام‌گذاری به `(Test1() می‌توانیم از قالب ()` استفاده کنیم:

```
void Test_SortAndFilterDocs() {
    ...
}
```

بسته به نوع پیچیدگی این تست، احتمالاً برای هر موقعیتی که دارد تست می‌شود یک تابع تست جداگانه در نظر می‌گیرید. شما می‌توانید از قالب `(Test_<FunctionName>_<Situation>()` استفاده کنید:

```
void Test_SortAndFilterDocs_BasicSorting() {
    ...
}

void Test_SortAndFilterDocs_NegativeValues() {
    ...
}
```

در اینجا از داشتن یک نام ناخوشایند و طولانی نترسید. این تابعی نیست که در کل کدپایه شما فراخوانی شود، بنابراین دلایل اجتناب از نام‌های طولانی برای تابع در اینجا اعمال نمی‌شود. نام تابع تست به طور موثر مانند یک کامنت عمل می‌کند. همچنین در صورت شکست خوردن تست، اکثر فریمورک‌ها نام تابعی که `assertion` در آن موفق نبوده است را چاپ خواهند کرد، بنابراین نام توصیفی، کمک زیادی خواهد کرد.

^۱ concatenate

توجه کنید که اگر از فریمورک تست استفاده می‌کنید، ممکن است قوانین یا قراردادهایی درباره نحوه نام‌گذاری متدها داشته باشند. به عنوان نمونه در مژول unittest در زبان Python انتظار می‌رود که نام متدها با کلمه «test» شروع شود.

هنگامی که صحبت از نام‌گذاری تابع کمکی در کد تست شما می‌شود، برجسته کردن اینکه آیا این تابع assertion‌هایی برای خودش نیز انجام می‌دهد و یا فقط یک «test-unaware» کمکی معمولی است، مفید خواهد بود.

به عنوان نمونه در این فصل هر تابع کمکی که assert() را صدا می‌زند به صورت Check... نام‌گذاری شده، اما تابع AddScoredDoc() فقط مانند یک تابع کمکی معمولی نام‌گذاری شده است.

مشکل این تست چیست؟

در ابتدای این فصل، گفتیم که حداقل هشت اشتباه در این تست وجود دارد:

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;
    SortAndFilterDocs(&docs);
    assert(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

اکنون که برخی از تکنیک‌های نوشتمن تست‌های بهتر را یاد گرفتیم، بباید آن‌ها را شناسایی کنیم:

۱. این تست بسیار طولانی و پر از جزئیات غیر مهم است. شما می‌توانید در یک جمله توصیف کنید که این کد چه کاری انجام می‌دهد، بنابراین دستور تست نباید خیلی طولانی باشد.
۲. افزودن تست جدید ساده نیست و شما وسوسه می‌شوید که از عملیات‌های `copy`، `paste` و `modify` استفاده کنید که این کار باعث طولانی‌تر شدن کد و همچنین پر از تکرار شود.
۳. پیام شکست خوردن تست خیلی مفید نیست. اگر این تست با شکست روبرو شود، تنها چیزی که خواهد گفت Assertion failed: `docs.size() == 3` است، که به شما اطلاعات کافی برای اشکال زدایی بیشتر را نمی‌دهد.
۴. این تست سعی می‌کند که چند چیز را در یک زمان تست کند، یعنی سعی می‌کند تست دو فیلتر منفی و عملیات مرتب سازی را به طور همزمان انجام دهد. اگر این موارد را در تست‌های چندگانه و جدا از هم انجام دهیم، کد تست خوانایی بیشتری خواهد داشت.
۵. ورودی‌های تست ساده نیستند. به طور خاص، امتیاز 99998.7- خیلی پر هیاهو! است و توجه شما را به خود جلب می‌کند، حتی اگر این مقدار خاص هیچ اهمیتی نداشته باشد. یک مقدار منفی ساده‌تر کفايت می‌کند.
۶. ورودی‌های تست به طور کامل با کد ارزیابی نمی‌شوند. به عنوان مثال زمانی که مقدار امتیاز برابر با 0 است را تست نمی‌کند.
۷. ورودی‌های دیگری مانند ورودی `vector` خالی، یک `vector` بسیار بزرگ یا امتیازات تکراری را تست نمی‌کند.
۸. نام `Test1()` بی معنی است. این اسم باید تابع یا شرایطی را که در حال تست شدن است را توصیف کند.

Test-Friendly توسعه

بعضی از کدها برای تست، ساده‌تر از برخی دیگر هستند. کد ایده‌آل برای تست، دارای یک `interface` است که به خوبی تعریف شده، وضعیت‌های زیاد یا تنظیمات دیگری نداشته و اطلاعات مخفی زیادی برای رسیدگی ندارد.

اگر کد خود را در حالی که می‌دانید بعداً برای آن تست خواهید نوشت، بنویسید یک چیز خنده دار اتفاق می‌افتد: شما به گونه‌ای که خود را طراحی می‌کنید تا تست آن آسان باشد! خوشبختانه، کدنویسی به این شیوه به این معنی است که شما در کل، کد بهتری تولید می‌کنید. به طور طبیعی

^۱ loud

طراحی‌های سازگار با تست، در اکثر موارد به کدی خوب و سازماندهی شده با قسمت‌های مجزا برای موارد جداگانه منتهی می‌شوند.

توسعه تست محور یا TEST-DRIVEN DEVELOPMENT

توسعه تست محور(TDD) یک سبک برنامه‌نویسی است که شما قبل از این که کد واقعی خود را بنویسید، تست‌ها را می‌نویسید. طرفداران TDD معتقدند که این فرآیند باعث می‌شود کیفیت کد، بسیار بیشتر از موقعی که تست‌ها را بعد از نوشتن کد می‌نویسید، بهبود یابد.

این موضوعی چالش برانگیز است و ما نمی‌خواهیم وارد آن شویم، ولی حداقل متوجه شدیم که تنها نگه‌داری تست‌ها در ذهن، در حین نوشتن کد، برای ایجاد کدی بهتر کمک می‌کند.

اما صرف نظر از اینکه TDD را به کار می‌برید یا نه، نتیجه پایانی داشتن کدی است که دیگر کدها را تست می‌کند. هدف این فصل این است که به شما کمک کند، تست‌های ساده‌تری برای خواندن و نوشتن ایجاد کنید.

در بین تمام راه‌های شکستن یک برنامه به کلاس‌ها و متدها، معمولاً جداشدنی‌ترین آن‌ها، ساده‌ترین گزینه برای تست هستند. از طرف دیگر، برنامه شما با تعداد زیادی فراخوانی متده در بین کلاس‌ها، همراه با تعداد زیادی پارامتر برای متدها، بسیار بهم پیوسته و پیچیده شده است که نه تنها فهمیدن کد برنامه را سخت، بلکه کد تست نیز زشت و خواندن و نوشتن آن دشوار خواهد شد.

داشتن تعداد زیادی کامپوننت خارجی(یعنی متغیرهای سراسری که باید مقداردهی شوند، کتابخانه‌ها یا فایل‌های پیکربندی^۱ که باید بارگیری^۲ شوند و غیره) نیز نوشتن تست‌ها را آزاردهنده‌تر^۳ می‌کند.

به طور کلی اگر در حال طراحی کد بوده و متوجه شدید که: این برای تست کردن یک کابوس خواهد بود! همین دلیل خوبی است که دست نگه داشته و مجدداً به طراحی فکر کنید. جدول ۱۴-۱ برخی از مشکلات تست‌های معمولی و طراحی آن‌ها را نشان می‌دهد.

^۱ config files

^۲ loaded

^۳ annoying

جدول ۱۴-۱. مشخصه‌های کد با تست‌پذیری پایین و مشکلات طراحی

مشکل طراحی	مشکل تست کردن	مشخصه
درک اینکه کدام توابع دارای چه اثرات جانبی است، سخت است. نمی‌توان درباره هر تابع به صورت جداگانه فکر کرد. برای درک اینکه آیا همه چیز کار می‌کند، باید کل برنامه را در نظر بگیرید.	همه وضعیت‌های سراسری باید برای هر تست ریست شوند (در غیر این صورت، تست‌های مختلف می‌توانند با یکدیگر تداخل داشته باشند).	استفاده از متغیرهای سراسری
هنگامی که یکی ازوابستگی‌ها ^۱ با شکست روپروردشود، سیستم به احتمال زیاد شکست خواهد خورد. درک اینکه این تغییر ممکن است چه اثراتی بگذارد، دشوارتر است. باسازی کردن کلاس‌ها کار سخت‌تری است. حالت‌های شکست سیستم بیشتر و بازیابی مسیر برای فکر کردن درباره آن سخت‌تر است.	نوشتن هرگونه تستی دشوار است، زیرا تعداد کارهای مقدماتی زیاد است و نوشتن تست‌ها کمتر سرگرم کننده‌اند، بنابراین افراد از نوشتن تست‌ها خودداری می‌کنند.	وابستگی کد به تعداد زیادی کامپوننت خارجی
این برنامه به احتمال زیاد دارای شبایط خاص (قابلیت) یا دیگر باگ‌های غیرقابل تکرار است. استدلال درباره این برنامه سخت است. ردیابی و رفع باگ‌ها در محصول بسیار دشوار خواهد بود.	تست‌ها شکننده و غیرقابل اعتماد هستند. تست‌هایی که گاهی شکست می‌خورند در نهایت نادیده گرفته می‌شوند.	رفتار غیرقطعی ^۲ کد

^۱ dependencies^۲ nondeterministic

در سوی دیگر، اگر یک طراحی داشته باشد که تست نوشتن برای آن ساده باشد، این یک نشانه خوب است. جدول ۱۴-۲ برخی از مزیت‌های تست و طراحی مشخصه‌ها را لیست کرده است.

جدول ۱۴-۲. مشخصه‌های کد با تست‌پذیری بالا و اینکه چگونه این شما را به سمت یک طراحی خوب هدایت می‌کند.

مزیت طراحی	مزیت تست‌پذیری	مشخصه
درک کلاس‌ها با وضعیت‌های کمتر، ساده‌تر و راحت‌تر است.	نوشتتن تست‌ها ساده‌تر است زیرا تنظیمات کمتری برای تست یک متند و حالت پنهان کمتری برای رسیدگی وجود دارد.	کلاس‌ها بدون وضعیت داخلی و یا دارای تعداد کمی از آن‌ها هستند.
کامپوننت‌های کوچک‌تر یا ساده‌تر قابلیت مازولاریتی بیشتری دارند و سیستم در کل، جداسازی بیشتری دارد.	برای تست کامل، test case کمتری نیاز است.	کلاس‌ها/تابع تنها یک کار انجام می‌دهند.
سیستم می‌تواند به شکل موازی توسعه داده شود، کلاس‌ها می‌توانند بدون مختل کردن بقیه سیستم، به راحتی تغییر کرده و یا حذف شوند.	هر کلاس می‌تواند به شکل مستقل تست شود(خیلی ساده‌تر از تست کردن چندین کلاس به طور هم زمان)	کلاس‌ها به کلاس‌های کمتری وابسته هستند(یعنی حداکثر جداسازی)
یادگیری و استفاده مجدد interface برای کدنویسان ساده‌تر است.	رفتارهایی که به خوبی تعریف شده‌اند، جهت تست وجود دارند.	تابع، interface‌های ساده‌ای دارند که به خوبی طراحی شده‌اند.

جزئیات بیشتر

این امکان پذیر است که تمرکز خیلی بیشتری روی تست شود. در اینجا چند مثال داریم:

- قربانی کردن خوانایی کد اصلی به دلیل فعل اعمال کردن تست‌ها. طراحی کد اصلی شما به صورت تست‌پذیر، باید دارای یک شرایط برد-برد باشد یعنی کد اصلی ساده‌تر و دارای جداسازی بیشتری است و تست‌ها نیز برای نوشتن راحت هستند. این کار اشتباهی است که تعداد زیادی اتصالات(پیچیدگی) در کد اصلی خود برای تست کردن آن‌ها اضافه کنید.

- در مورد پوشش ۱۰۰ درصدی تست وسوساً داشته باشد. تست ۹۰ درصد ابتدای کد، اغلب کمتر از تست ۱۰ درصد آخر وقت گیر است. آن ۱۰ درصد احتمالاً شامل رابط کاربری^۲ یا موارد خطای گنگ می‌شود، یعنی مواردی که هزینه باگ در آن‌ها خیلی زیاد نیست و

^۱ plumbing

^۲ user interface

ارزشی برای تست کردن ندارد. اما حقیقت این است که شما هرگز پوشش ۱۰۰ درصدی، نخواهید داشت.

- اگر یک باگ هم از دست نرفته است، احتمالاً یک ویژگی از دست رفته یا شما متوجه نشده‌اید که آن مشخصه باید تغییر می‌کرده است. بسته به اینکه باگ‌های شما چقدر هزینه بر هستند، این مهم است که چقدر از زمان توسعه را برای تست کردن کد هزینه کنید. اگر در حال ساخت یک نمونه اولیه وبسایت هستید، احتمالاً به هیچ وجه ارزش نوشتمن کد تست را نداشته باشد. از طرف دیگر، اگر شما یک کنترل کننده برای یک سفینه فضایی یا وسیله پژوهشی نوشتیده‌اید، بی شک مرکز اصلی شما باید تست کردن آن باشد.
- اجازه دهید انجام تست در مسیر توسعه محصول باشد. ما با موقعیت‌هایی روبرو شده‌ایم که به جای آن که تست، فقط یکی از جنبه‌های پروژه باشد، به کل پروژه حاکم شده است. گاهی اوقات تشریفات عملیات تست بیش از حد مورد توجه قرار می‌گیرد و کدنویسان متوجه نمی‌شوند که می‌توانند وقت گرانبهایشان را درجای بهتری صرف کنند.

خلاصه فصل

در کد تست، خوانایی خیلی مهم است. اگر تست‌های شما خیلی خوانا باشند، به نوبه خود بسیار قابل نوشتمن خواهند بود و به همین دلیل تعداد افراد بیشتری از آن‌ها استفاده و به آن‌ها تست‌های جدید اضافه می‌کنند. همچنین، اگر کد اصلی را برای تست، ساده طراحی کرده‌اید، در کل، کد شما طراحی بهتری خواهد داشت.

در اینجا نکات خاصی در مورد اینکه چگونه تست‌های خود را بهبود دهید، داریم:

- سطح بالای هر تست باید تا حد امکان مختصر باشد. در حالت ایده‌آل، هر ورودی/خروجی تست می‌تواند تنها در یک خط از کد شرح داده شود.
- اگر تست شما با شکست مواجه شد، باید پیام خطای منتشر کند که باعث شود ردیابی و تصحیح باگ ساده باشد.
- از ساده‌ترین ورودی‌های تست که به طور کامل کد شما را ارزیابی می‌کنند استفاده کنید.
- به توابع تست خود یک اسم کاملاً توصیفی بدهید تا مشخص باشد هر کدام از آن‌ها چه چیزی را تست می‌کند. به جای `Test10` از نامی مانند `Test_<FunctionName>_<Situation>` استفاده کنید.
- و مهم‌تر از همه اینکه، اصلاح و اضافه کردن تست‌های جدید را آسان کنید.

فصل پانزدهم

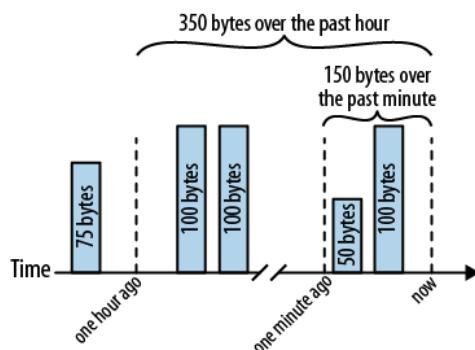
طراحی و پیاده سازی یک «شمارنده دقیقه/ساعت»



باید به ساختار داده استفاده شده در یک کد محصول واقعی نگاهی بیندازیم؛ یک «شمارنده دقیقه/ساعت». ما شما را با فرآیند طبیعی تفکر یک مهندس، آشنا می‌کنیم، ابتدا سعی در حل این مسئله داشته و سپس به بهبود کارابی و افزودن ویژگی‌هایی به آن خواهیم پرداخت، از همه مهمتر، سعی خواهیم کرد با استفاده از اصولی که در این کتاب بیان شده است، کد را برای خواندن ساده نگه داریم. البته ممکن است در این مسیر دچار چند اشتباہ شویم. ببینید آیا می‌توانید آن‌ها را دنبال کرده و تشخیص دهید؟

مسئله

ما می‌خواهیم تعداد byte که در یک دقیقه و همچنین یک ساعت گذشته در یک سرور وب منتقل شده است را ردیابی کنیم. در اینجا تصویری از چگونگی نگه‌داری آن‌ها داریم:



در ابتدا به نظر می‌رسد که این یک مسئله نسبتاً راحت است، اما همان گونه که خواهید دید، حل کردن آن به صورت کارآمد یک چالش جالب خواهد بود. باید با تعریف رابط کلاس شروع کنیم.

تعریف Interface کلاس

اولین نسخه از Interface کلاس ما به زبان C++ به صورت زیر است:

```
class MinuteHourCounter {
public:
    // Add a count
    void Count(int num_bytes);
    // Return the count over this minute
    int MinuteCount();
    // Return the count over this hour
    int HourCount();
};
```

قیل از پیاده سازی این کلاس، اجازه دهید از طریق مشاهده نامها و کامنت‌ها، ببینیم چیزی وجود دارد که بخواهیم آن را تغییر دهیم یا نه؟

بهبود نام‌ها

نام کلاس MinuteHourCounter خیلی خوب است. این نام خیلی خاص، مشخص و هنگام تلفظ آسان است. با توجه به نام کلاس، نام متدهای MinuteCount() و HourCount() نیز منطقی هستند. ممکن است شما آن‌ها را GetHourCount() و GetMinuteCount() بنامید اما این کار هیچ کمکی نمی‌کند. همان گونه که در فصل ۳ بیان کردیم، نام‌ها نباید گمراه کننده یا موجب برداشت‌های متناقض باشند، در حالی که کلمه get برای خیلی از افراد اشاره به «دسترسی سریع و سبک^۱» دارد و همان گونه که خواهید دید، پیاده سازی این مسئله، سبک نخواهد بود، بنابراین بهتر است از get در نام‌گذاری استفاده نکنیم.

ظاهرا نام متدهای Count() گیج کننده است. ما از همکاران خود سوال کردیم: به نظرتان() چه کاری، انجام می‌دهد؟ عده‌ای تصویر می‌کردند که این یعنی تعداد کل شمارش‌ها در کل زمان را بر می‌گرداند. این اسم کمی متناقض^۲ بوده و بدون هدف در نظر گرفته شده است. مشکل این که (در زبان انگلیسی) Count هم اسم و هم فعل است و حتی می‌تواند به این معنی باشد که «من یک شمارش از تعداد نمونه‌هایی که تا کنون دیده‌اید را می‌خواهم» یا «من از شما می‌خواهم که این نمونه را بشمارید».

در اینجا نام‌های متناظر برای جایگزین شدن با Count آورده شده است:

- Increment()
- Observe()
- Record()
- Add()

هم گمراه کننده است زیرا به این معنی است که مقداری وجود دارد که فقط افزایش (Increment) می‌یابد. (در مثال ما، شمارش ساعت به صورت چرخشی است).

Observe() خوب است اما کمی مبهم است.

Record() نیز مشکل کلمه/فعل بودن را دارد، بنابراین گزینه خوبی نیست.

^۱ lightweight accessor

^۲ counterintuitive

Add() جالب است زیرا می‌تواند به معنای «این را به شکل عددی اضافه کن» یا «اضافه کردن به یک لیست از داده» باشد. در مسئله‌ما، مقداری از مفهوم هر دو جمله وجود دارد، در نتیجه می‌تواند به کار آید.

بنابراین ما متدهای خود را به void Add(int num_bytes) تغییر نام می‌دهیم. اما نام آرگومان num_bytes خیلی خاص است. بله، مورد کاربرد اصلی ما برای شمارش byte‌ها است، اما نیازی نیست که MinuteHourCounter آن را بداند. ممکن است شخص دیگری از این کلاس برای شمارش کوئری‌ها یا تعاملات پایگاه داده استفاده کند. ما می‌توانیم از نام عمومی‌تری مانند delta استفاده کنیم، اما عبارت delta اغلب برای کاربردهایی استفاده می‌شود که مقدار آن می‌تواند منفی باشد که ما چنین چیزی را نمی‌خواهیم. نام count باید بهتر باشد. این نام ساده و عمومی است و اشاره بر غیرمنفی بودن دارد. همچنین به ما اجازه می‌دهد که کاربر برداشت دوپهلوی کمتری از کلمه count داشته باشد.

بهبود کامنت‌ها

تا اینجا این رابط کلاس را داریم:

```
class MinuteHourCounter {
public:
    // Add a count
    void Add(int count);
    // Return the count over this minute
    int MinuteCount();
    // Return the count over this hour
    int HourCount();
};
```

بیایید هر یک از این متدها را کامنت گذاری کنیم و آن‌ها را بهبود دهیم. اولین مورد را در نظر بگیرید:

```
// Add a count
void Add(int count);
```

این کامنت به این شکل کاملاً زائد است. یا باید حذف شود و یا اینکه بهبود یابد. یک نسخه بهبود یافته آن به صورت زیر است:

```
// Add a new data point (count >= 0).
// For the next minute, MinuteCount() will be Larger by +count.
// For the next hour, HourCount() will be larger by +count.
void Add(int count);
```

اکنون بباید کامنت مربوط به `MinuteCount()` را در نظر بگیریم:

```
// Return the count over this minute
int MinuteCount();
```

وقتی که از همکاران خود درباره معنای این کامنت سوال کردیم، دو تفسیر متناقض بیان کردند:

۱. برگرداندن شمارش در طول دقیقه فعلی مثلا 12:13pm
۲. برگرداندن شمارش در طول ۶۰ ثانیه گذشته، بدون در نظر گرفتن مرزهای دقیقه و ساعت.

تعابیر دوم چیزی است که در واقع عمل می‌کند. بنابراین بباید این ابهام را با زبانی دقیق‌تر و با جزئیات بیشتر پاکسازی کنیم:

```
// Return the accumulated count over the past 60 seconds.
int MinuteCount();
```

به طور مشابه، ما باید کامنت مربوط به `HourCount()` را نیز بهبود دهیم.

```
// Track the cumulative counts over the past minute and over the past hour.

// Useful, for example, to track recent bandwidth usage.

class MinuteHourCounter {
    // Add a new data point (count >= 0).

    // For the next minute, MinuteCount() will be Larger by +count.

    // For the next hour, HourCount() will be larger by +count.

    void Add(int count);

    // Return the accumulated count over the past 60 seconds.

    int MinuteCount();
```

```
// Return the accumulated count over the past 3600 seconds.
int HourCount();
};


```

برای کوتاهتر بودن کدها، ما از این پس کامنت‌ها را از لیست کدها خارج خواهیم کرد.

به دست آوردن یک چشم انداز بیرونی

احتمالاً می‌دانید که قبلاً مواردی وجود داشت که ما آن‌ها را از همکاران خود درخواست می‌کردیم. درخواست یک چشم انداز بیرونی یک راه عالی جهت تست این مورد است که آیا کد شما کاربر-پسند^۱ است یا نه؟ سعی کنید که در اولین برداشت‌های همکارانتان راحت باشید، زیرا افراد دیگر ممکن است به همان نتایج برسند و احتمالاً شما نیز شش ماه بعد جزو همان «افراد دیگر» باشید.

تلash شماره ۱: یک راه حل ساده و بی تکلف

بیایید از این بحث بگزیریم و به سراغ حل مسئله برویم. ما با یک راه حل مستقیم شروع می‌کنیم: فقط لیستی از برچسب زمانی^۲ Event‌ها را نگهداری کنید.

```
class MinuteHourCounter {
    struct Event {
        Event(int count, time_t time) : count(count), time(time) {}
        int count;
        time_t time;
    };
    list<Event> events;
public:
    void Add(int count) {
        events.push_back(Event(count, time()));
    }
    ...
};


```

سپس می‌توانیم جدیدترین رخدادها را در صورت لزوم محاسبه کنیم:

```
class MinuteHourCounter {
```

^۱ user-friendly
^۲ timestamped

```

...
int MinuteCount() {
    int count = 0;
    const time_t now_secs = time();
    for (list<Event>::reverse_iterator i = events.rbegin();
        i != events.rend() && i->time > now_secs - 60; ++i) {
        count += i->count;
    }
    return count;
}
int HourCount() {
    int count = 0;
    const time_t now_secs = time();
    for (list<Event>::reverse_iterator i = events.rbegin();
        i != events.rend() && i->time > now_secs - 3600; ++i) {
        count += i->count;
    }
    return count;
}
};

```

آیا درک این کد آسان است؟

اگرچه این راه حل صحیح است، اما خالی از اشکالات خوانایی نیست:

- حلقه‌های `for` کمی سنگین و پیچیده هستند. اکثر افراد هنگام خواندن این بخش از کد، به طور قابل توجهی از سرعت خواندنشان کم می‌شود (آن‌ها حداقل باید اطمینان حاصل کنند که اشکالی وجود ندارد).
- تقریباً `HourCount()` و `MinuteCount()` یکسان هستند. اگر بتوانند کدهای تکرار شده را به اشتراک بگذارند، باعث می‌شود این کد کوچک‌تر شود. به طور خاص این جزئیات مهم‌اند، زیرا افزونگی کد نسبتاً پیچیده است (بهتر است همه کدهای دشوار را در یک مکان محدود کنید).

نسخه ساده‌تر از نظر خوانایی

کد مربوط به HourCount() و MinuteCount() تنها در یک ثابت متفاوت هستند (یعنی 60 در مقابل 3600). واضح است که بازسازی آن‌ها به این صورت خواهد بود که یک متدهای کمکی برای مدیریت هر دو آن‌ها معرفی کنیم:

```
class MinuteHourCounter {
    list<Event> events;
    int CountSince(time_t cutoff) {
        int count = 0;
        for (list<Event>::reverse_iterator rit = events.rbegin();
             rit != events.rend(); ++rit) {
            if (rit->time <= cutoff) {
                break;
            }
            count += rit->count;
        }
        return count;
    }
public:
    void Add(int count) {
        events.push_back(Event(count, time()));
    }
    int MinuteCount() {
        return CountSince(time() - 60);
    }
    int HourCount() {
        return CountSince(time() - 3600);
    }
};
```

موارد قابل اشاره در کد جدید عبارتند از:

اول، توجه کنید که CountSince() یک پارامتر دقیق cutoff را به جای یک مقدار نسبی secs_ago دریافت می‌کند (یعنی 60 یا 3600). هر چند در هر صورت کار خواهد کرد، اما شیوه CountSince() کار ساده‌تری انجام می‌دهد.

دوم، ما متغیر تکرار شونده را از `i` به تغییر نام دادیم. نام `i` معمولا برای شاخصهای عدد صحیح استفاده می‌شود. ما در استفاده از آن نام که معمولا برای تکرار کننده‌ها است اندیشیدیم، اما در این مورد یک تکرار کننده معکوس داریم و این واقعیت، برای صحت کد بسیار مهم است. با داشتن یک نام متغیر که با پیشوند `r` شروع می‌شود، یک فرینه سازی ساده را برای دستوراتی شبیه `events.render()`!`rit` اضافه می‌کنیم.

در نهایت، ما شرط `rit->time <= cutoff` را از حلقه `for` استخراج کرده و آن را با یک دستور `if` جداگانه ساختیم. دلیل این کار این بود که حلقه‌های سنتی `for` به شکل `(begin; end; advance)` به شکل (آغاز؛ پایان؛ پیشرفت) خواندن می‌شوند. خواننده می‌تواند بلافاصله آن را درک کند که به معنی «همه عناصر را پشت سر بگذار» است و به تفکر بیشتری نیاز ندارد.

مشکلات کارایی

اگرچه ما ظاهر کد را بهبود دادیم، ولی این طراحی دو مشکل جدی در کارایی دارد:

۱. فقط در حال رشد کردن است.

این کلاس همه رخدادهایی که تا کنون دیده شده را نگهداری می‌کند. این کار یک مقدار بی حد و حصر از حافظه را استفاده می‌کند! در حالت ایده‌آل، `MinuteHourCounter` باید به صورت خودکار رخدادهایی که قدیمی‌تر از یک ساعت هستند را حذف کند، زیرا دیگر به آن‌ها نیازی نیست.

۲. `HourCount()` و `MinuteCount()` خیلی کند هستند.

متدهای `CountSince()` دارای سرعتی برابر $O(n)$ است که `n` تعداد نقاط داده^۱ در پنجره زمانی مربوطه است. تصویر کنید یک سرور با حداکثر کارایی، تابع `Add()` را هزاران مرتبه در هر ثانیه فراخوانی کند. هر فراخوانی `HourCount()` باید از طریق یک میلیون نقاط داده محاسبه شود! در حالت ایده‌آل، `MinuteHourCounter` باید متغیرهای `hour_count` و `minute_count` را به صورت جداگانه به گونه‌ای نگه داری کند که با هر فراخوانی برای `Add()` به روزرسانی شده باشند.

^۱ data points

تلاش شماره ۲: طراحی تسمه نقاله^۱

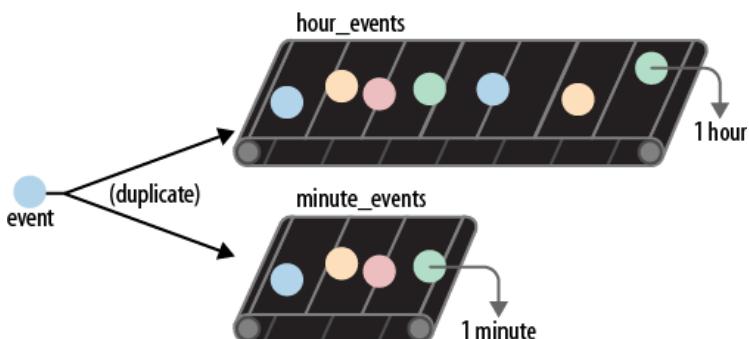
ما به یک طراحی نیاز داریم که این دو مشکل را حل کند:

۱. حذف داده‌ای که ما دیگر به آن نیازی نداریم.
۲. نگه‌داری به روز رسانی‌های مجموع hour_count و minute_count که از قبل محاسبه شده‌اند.

در اینجا نحوه انجام این کار را نشان داده‌ایم:

ما از لیست خود مانند یک تسمه نقاله استفاده می‌کنیم یعنی زمانی که داده جدید به یک انتهای می‌رسد، آن را به مجموع خود اضافه کرده و زمانی که داده خیلی قدیمی باشد، آن داده از انتهای دیگر «سقوط کرده» و ما آن را از مجموع خود کم می‌کنیم.

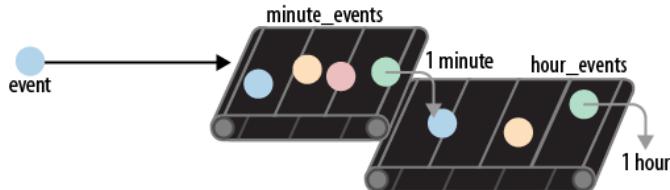
چندین راه برای پیاده سازی این طراحی تسمه نقاله وجود دارد. یک راه این است که دو لیست مستقل نگه‌داری کنید، یکی برای رخدادها در یک دقیقه گذشته و یکی هم برای رخدادها در یک ساعت گذشته. حال وقتی که یک رخداد جدید وارد شد، یک کپی به هر دو لیست اضافه کنید.



این روش بسیار ساده، اما ناکارآمد است زیرا از هر رخداد دو کپی ایجاد می‌کند.

راه دیگر این است که دو لیست نگه‌داری کنیم که ابتدا رخدادها وارد لیست اول شده («رخدادهای یک دقیقه آخر») و سپس آن‌ها وارد لیست دوم شوند (رخدادهای یک ساعت آخر اما نه «رخدادهای یک دقیقه آخر»).

^۱ Conveyor Belt



این طراحی تسمه نقاله دو مرحله‌ای به نظر کارآیی بیشتری دارد، بنابراین اجازه دهد این را پیاده سازی کنیم.

پیاده سازی طراحی تسمه نقاله دو مرحله‌ای

بیایید با لیست کردن همه اعضای کلاس خود، شروع کنیم:

```
class MinuteHourCounter {
    list<Event> minute_events;
    list<Event> hour_events; // only contains elements NOT in minute_events
    int minute_count;
    int hour_count; // counts ALL events over past hour, including past minute
};
```

معمای طراحی تسمه نقاله این است که می‌تواند با گذشت زمان رخدادها را جا به جا^۱ کند، بنابراین رخدادها از hour_events به minute_events حرکت داده شده و minute_count و hour_count به ترتیب به روزرسانی می‌شوند. برای انجام این کار ما یک متدهای ShiftOldEvents() به نام() ترتیب خواهیم ساخت. هنگامی که این متدهای را داشته باشیم، پیاده سازی بقیه کلاس آسان است:

```
void Add(int count) {
    const time_t now_secs = time();
    ShiftOldEvents(now_secs);
    // Feed into the minute list (not into the hour list--that will happen later)
    minute_events.push_back(Event(count, now_secs));
    minute_count += count;
    hour_count += count;
}

int MinuteCount() {
    ShiftOldEvents(time());
    return minute_count;
```

^۱ shift

```

}

int HourCount() {
    ShiftOldEvents(time());
    return hour_count;
}

```

به طور واضح، ما همه کارهای کثیف را در تابع ShiftOldEvents() به تعویق انداختیم:

```

//Find and delete old events, and decrease hour_count and minute_count accordingly.
void ShiftOldEvents(time_t now_secs) {
    const int minute_ago = now_secs - 60;
    const int hour_ago = now_secs - 3600;

    // Move events more than one minute old from 'minute_events' into 'hour_events'
    // (Events older than one hour will be removed in the second loop.)
    while (!minute_events.empty() && minute_events.front().time <= minute_ago) {
        hour_events.push_back(minute_events.front());
        minute_count -= minute_events.front().count;
        minute_events.pop_front();
    }

    // Remove events more than one hour old from 'hour_events'
    while (!hour_events.empty() && hour_events.front().time <= hour_ago) {
        hour_count -= hour_events.front().count;
        hour_events.pop_front();
    }
}

```

آیا کار ما تمام شده است؟

ما دو مشکل کارآیی را که قبلا اشاره شده بود حل کردیم و راه حل ما به خوبی کار می‌کند. برای بسیاری از برنامه‌ها، این راه حل به اندازه کافی مناسب خواهد بود. اما همچنان برخی از نواقص باقی مانده‌اند.

اول اینکه، طراحی اصلا انعطاف‌پذیر نیست. فرض کنید ما بخواهیم شمارش بیش از ۲۴ ساعت گذشته را نگه‌داری کنیم. این امر مستلزم ایجاد تغییرات زیادی در کد است. همان‌گونه که احتمالا متوجه شده‌اید، (ShiftOldEvents() یک تابع بسیار متراکم با تعامل دقیق بین داده‌های دقیقه و ساعت است.

دوم، این کلاس دارای مقدار حافظه بسیار بزرگ است. فرض کنید شما یک سرور با ترافیک خیلی زیاد کهتابع() را ۱۰۰ بار در هر ثانیه فراخوانی می‌کند دارید. از آنجا که ما برای همه داده‌ها در یک ساعت گذشته صبر می‌کنیم، این کد در نهایت باید به حدود ۵MB حافظه نیاز داشته باشد.

به طور کلی، هرچه Add() بیشتر فراخوانی شده باشد، ما حافظه بیشتری استفاده می‌کنیم. در یک محیط تولید محصول، کتابخانه‌هایی که از مقدار حافظه زیادی به صورت پیش‌بینی نشده استفاده می‌کنند، خوب نیستند. در حالت ایده‌آل، MinuteHourCounter باید مقدار ثابتی از حافظه را بدون توجه به تعداد فراخوانی‌های Add() استفاده کند.

تلاش شماره ۳: طراحی یک Time-Bucketed

شاید متوجه نشده باشید، اما هر دو پیاده سازی قبلی یک باگ کوچک داشتند. ما از time_t برای ذخیره‌سازی برچسب زمانی^۱ استفاده کردیم، که تعداد صحیح اعداد از ثانیه‌ها را ذخیره می‌کرد. بخارط این گرد کردن، در واقع MinuteCount()، بسته به اینکه دقیقاً چه زمانی آن را صدا می‌زنید، چیزی بین ۵۹ و ۶۰ ثانیه را برابر می‌گرداند.

به عنوان مثال، اگر یک رخداد در $time = 0.99$ اتفاق افتد، این زمان به ثانیه $t=0$ گرد خواهد شد و اگر شما MinuteCount() را در ثانیه $time = 60.1$ صدا بزنید، این تابع، مقدار مجموع را برای رخدادهای $t=1,2,3,\dots,60$ برخواهد گرداند. بنابراین اولین رخداد، از دست خواهد رفت، حتی اگر از نظر فنی کمتر از یک دقیقه قبیل باشد.

به طور متوسط، داده‌ای به ارزش ۵۹.۵ ثانیه و HourCount() داده‌ای به ارزش ۳۵۹۹.۵ را برابر می‌گرداند(یک خطای ناچیز).

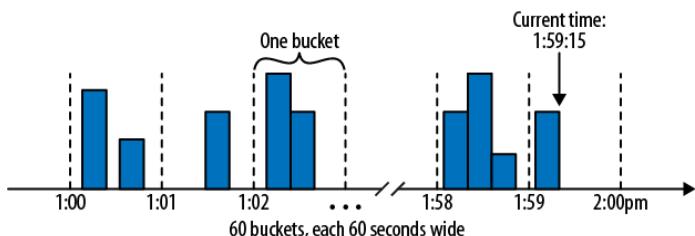
ما می‌توانیم همه این‌ها را با استفاده از یک زمان با granularity^۲ فرعی تصحیح کنیم. اما جالب این است که، اکثر برنامه‌هایی که از یک MinuteHourCounter استفاده می‌کنند در درجه اول به این میزان از دقت نیازی ندارند. ما از این حقیقت برای طراحی جدیدی از MinuteHourCounter

^۱ Timestamp

^۲ دانه‌بندی: به تقسیم یک سیستم به ریزترین و جزئی‌ترین اجزای آن می‌گویند. گاهی می‌توان جزئیات سیستم یا مشخصات آن را نیز دانه بندی کرد. این اصطلاح مثالی است از تقسیم بندی یک حیاط خلوت به دانه‌های موجود در آن نسبت به تقسیم بندی آن به مقیاس فوت.

بهره‌برداری خواهیم کرد که خیلی سریعتر و از فضای کمتری استفاده می‌کند. این یک trade-off^۱ و کارآیی^۲ است که ارزش پرداختن به آن را دارد.

ایده اصلی این است که همه رخدادها را داخل یک پنجره زمانی کوچک در یک سطح^۳ قرار داده و آن رخدادها را با یک مقدار مجموع تکی خلاصه کنیم. به عنوان نمونه، رخدادهای یک دقیقه گذشته را می‌توان داخل ۶۰ سطح مجزا، با عرض یک ثانیه و رخدادهای یک ساعت گذشته را در ۶۰ سطح مجزا، با عرض ۱ دقیقه‌ای، قرار داد.



همانطور که در شکل نشان داده شده است، با استفاده از سطلهای متعدد MinuteCount() و HourCount() به دقت ۱ قسمت در هر ۶۰ قسمت رسیده‌اند که قابل قبول است.^۴

در صورتی که برای حافظه با مقدار بزرگتر، به دقت بیشتری نیاز دارید، می‌توان از سطلهای بیشتری برای تبادلات استفاده کرد. اما مهمترین چیز این است که این طراحی دارای یک مصرف حافظه قابل پیش‌بینی و تصحیح شده است.

پیاده سازی طراحی Time-Bucketed

پیاده سازی این طراحی فقط با یک کلاس، تعداد زیادی کد پیچیده ایجاد می‌کند که مرور کردن آن‌ها در ذهن سخت خواهد بود. در عوض، ما توصیه خود را از فصل ۱۱ دنبال می‌کنیم، یعنی «یک وظیفه در یک زمان» و کلاس‌های جداگانه‌ای برای مدیریت بخش‌های مختلف این مسئله ایجاد می‌کنیم.

^۱ Precision

^۲ Performance

^۳ Bucket

^۴ مشابه راه حل‌های قبلی آخرین سطل به طور متوسط فقط نصف آن پر خواهد بود. با این طراحی، ما می‌توانیم با نگه‌داری ۶۰ سطل به جای ۶ سطل و نادیده گرفتن سطل در حال پر شدن فعلی، تخمین کم (underestimate) را اصلاح کنیم. اما این باعث می‌شود که داده‌ها تا حدی قدیمی باشند. یک اصلاح بهتر این است که سطل درحال پر شدن را با یکتابع مکمل از قدیمی‌ترین سطل، برای بدست آوردن یک شمارش بدون تبعیض و به روز، ترکیب کنیم. این پیاده سازی را به عنوان یک تمرین به عهده خواننده می‌گذاریم.

برای شروع، بباید یک کلاس جداگانه برای پیگیری شمارش‌ها برای یک مدت زمان واحد (مثلاً حداقل یک ساعت) ایجاد کنیم. ما نام این کلاس را TrailingBucketCounter می‌گاریم. این در اصل یک نسخه عمومی از MinuteHourCounter است که تنها یک واحد زمانی را مدیریت می‌کند. به شکل زیر خواهد بود:

```
// A class that keeps counts for the past N buckets of time.
class TrailingBucketCounter {
public:
    // Example: TrailingBucketCounter(30, 60) tracks the last 30 minute-
    // buckets of time.
    TrailingBucketCounter(int num_buckets, int secs_per_bucket);
    void Add(int count, time_t now);
    // Return the total count over the last num_buckets worth of time
    int TrailingCount(time_t now);
};
```

احتمالاً نگرانید که چرا Add() و TrailingCount() به زمان فعلی (time_t) به عنوان یک آرگومان نیاز دارند. نگران نباشید! اگر این متدها فقط time فعلی خودشان را محاسبه می‌کردند، ساده‌تر نبود؟

اگرچه ممکن است عجیب به نظر برسد، صرف نظر کردن از زمان فعلی، چندین مزیت دارد. اول اینکه TrailingBucketCounter که یک کلاس بدون زمان است را ایجاد می‌کند، که برای تست ساده‌تر بوده و کمتر مستعد اشکال^۱ است. دوم اینکه، همه فراخوانی‌ها را برای time در MinuteHourCounter نگه‌داری می‌کند. داشتن یک سیستم حساس به زمان، در صورتی که به شما کمک کند همه فراخوانی‌های time را در یک مکان به دست آورید، مفید است.

با فرض اینکه TrailingBucketCounter قبل از تعریف شده، پیاده سازی شده باشد، ساده خواهد بود:

```
class MinuteHourCounter {
    TrailingBucketCounter minute_counts;
    TrailingBucketCounter hour_counts;
public:
    MinuteHourCounter() :
        minute_counts(/* num_buckets = */ 60, /* secs_per_bucket = */ 1),
        hour_counts( /* num_buckets = */ 60, /* secs_per_bucket = */ 60) {
```

^۱ bug-prone

```

    }

void Add(int count) {
    time_t now = time();
    minute_counts.Add(count, now);
    hour_counts.Add(count, now);
}

int MinuteCount() {
    time_t now = time();
    return minute_counts.TrailingCount(now);
}

int HourCount() {
    time_t now = time();
    return hour_counts.TrailingCount(now);
}

};

}

```

خوانایی این کد خیلی بیشتر بوده و اعطاف پذیری بیشتری نیز دارد. این کار ساده‌ای است که تعداد سطلهای را، جهت بهبود دقت افزایش دهیم، اما به یاد داشته باشید که مصرف حافظه افزایش پیدا می‌کند.

پیاده سازی TrailingBucketCounter

اکنون تنها چیزی که برای پیاده سازی باقی مانده، کلاس TrailingBucketCounter است. یک بار دیگر، ما قصد داریم یک کلاس کمکی برای جداسازی مشکل بعدی ایجاد کنیم. یک ساختار داده به نام ConveyorQueue ایجاد خواهیم کرد که وظیفه‌اش سر و کار داشتن با شمارش‌ها و مجموع آن‌ها است. کلاس TrailingBucketCounter می‌تواند روی کار جابجا کردن ConveyorQueue مطابق با اینکه چه مقدار زمان سپری شده است، تمرکز کند.

در ادامه رابط ConveyorQueue را مشاهده می‌کنید:

```

// A queue with a maximum number of slots, where old data "falls off" the end.

class ConveyorQueue {
    ConveyorQueue(int max_items);

    // Increment the value at the back of the queue.
    void AddToBack(int count);

    // Each value in the queue is shifted forward by 'num_shifted'.
    // New items are initialized to 0.

    // Oldest items will be removed so there are <= max_items.
}

```

```

void Shift(int num_shifted);
    // Return the total value of all items currently in the queue.
int TotalSum();
};

```

فرض کنید این کلاس پیاده سازی شده بود، نگاه کنید که پیاده سازی TrailingBucketCounter چقدر آسان است:

```

class TrailingBucketCounter {
    ConveyorQueue buckets;
    const int secs_per_bucket;
    time_t last_update_time; // the last time Update() was called
    // Calculate how many buckets of time have passed and Shift() accordingly.
    void Update(time_t now) {
        int current_bucket = now / secs_per_bucket;
        int last_update_bucket = last_update_time / secs_per_bucket;

        buckets.Shift(current_bucket - last_update_bucket);
        last_update_time = now;
    }
public:
    TrailingBucketCounter(int num_buckets, int secs_per_bucket) :
        buckets(num_buckets),
        secs_per_bucket(secs_per_bucket) {
    }
    void Add(int count, time_t now) {
        Update(now);
        buckets.AddToBack(count);
    }
    int TrailingCount(time_t now) {
        Update(now);
        return buckets.TotalSum();
    }
};

```

کار شکستن کد به دو کلاس ConveyorQueue و TrailingBucketCounter نمونه دیگری از بحثی است که در فصل ۱۱ داشتیم، یعنی یک وظیفه در یک زمان. همچنین می‌توانستیم بدون

ConveyorQueue و با پیاده سازی مستقیم همه چیز داخل TrailingBucketCounter، کار مد نظر خود را انجام دهیم. اما با شکستن کد به دو کلاس، هضم کد آسان‌تر است.

پیاده سازی ConveyorQueue

حال تنها چیزی که باقی مانده است پیاده سازی کلاس ConveyorQueue است:

```
// A queue with a maximum number of slots, where old data gets shifted off the end.
class ConveyorQueue {
    queue<int> q;
    int max_items;
    int total_sum; // sum of all items in q
public:
    ConveyorQueue(int max_items) : max_items(max_items), total_sum(0) {
    }
    int TotalSum() {
        return total_sum;
    }
    void Shift(int num_shifted) {
        // In case too many items shifted, just clear the queue.
        if (num_shifted >= max_items) {
            q = queue<int>(); // clear the queue
            total_sum = 0;
            return;
        }
        // Push all the needed zeros.
        while (num_shifted > 0) {
            q.push(0);
            num_shifted--;
        }
        // Let all the excess items fall off.
        while (q.size() > max_items) {
            total_sum -= q.front();
            q.pop();
        }
    }
    void AddToBack(int count) {
        if (q.empty()) Shift(1); // Make sure q has at least 1 item.
        q.back() += count;
    }
}
```

```

    total_sum += count;
}
};

```

اکنون کار ما تمام شده است! ما یک MinuteHourCounter داریم که سریعتر و از نظر مصرف حافظه کارآیی بالاتری دارد، علاوه بر این، TrailingBucketCounter انعطاف‌پذیرتر است و در نتیجه به راحتی قابل استفاده مجدد است. به عنوان نمونه خیلی ساده خواهد بود که یک RecentCounter تطبیق پذیرتر^۱ که بتواند دامنه وسیعی از فاصله‌ها^۲ را شمارش کند، ساخته شود (مانند آخرین روز یا آخرین ده دقیقه).

مقایسه سه راه حل

بیایید راه حل‌هایی که در این فصل ارائه شد را با هم مقایسه کنیم. جدول زیر اندازه کد و آمار کارآیی را با فرض یک مورد کاربرد ترافیک بالا از ۱۰۰ مرتبه استفاده از تابع `Add()` در هر ثانیه (۱۰۰ sec) نشان می‌دهد:

خطا در <code>HourCount()</code>	صرف حافظه	هزینه به ازای هر <code>HourCount()</code>	تعداد خطوط کد	راه حل
1 part per 3600	unbounded	$O(\#events-per-hour)$ (~3.6 million)	33	Naive solution
1 part per 3600	$O(\#events-per-hour)$ (~5MB)	$O(1)$	55	Conveyor belt design
1 part per 60	$O(\#buckets)$ (~500 bytes)	$O(1)$	98	Time-bucketed design (60 buckets)

توجه کنید که مقدار کل کد، برای راه حل‌های نهایی سه کلاس بیشتر از هر یک از موارد دیگر است. با این حال کارآیی بسیار بهتر و طراحی آن انعطاف‌پذیری بیشتری خواهد داشت. همچنین، خواندن هر کلاس به صورت جداگانه راحت‌تر است. این همیشه یک تغییر مثبت است: داشتن ۱۰۰ خط که همه آن‌ها برای خواندن ساده هستند، بهتر از ۵۰ خطی است که خواندن شان ساده نیست. گاهی اوقات، شکستن یک مشکل به چندین کلاس می‌تواند پیچیدگی بین کلاسی ایجاد کند (در صورتی که راه حل تک کلاسی چنین کاری نمی‌کند). در این حالت، یک زنجیره خطی ساده برای عبور از یک کلاس به کلاس بعدی وجود دارد و فقط یکی از کلاس‌ها در معرض دید کاربر نهایی قرار می‌گیرد. به طور کلی مزیت شکستن این مسئله به زیرمسئله‌ها باعث یک پیروزی است.

^۱ versatile

^۲ intervals

خلاصه فصل

بیایید مراحلی را که برای رسیدن به طرح نهایی MinuteHourCounter طی کردیم مرور کنیم. این فرآیند از چگونگی تکامل قسمت‌های مختلف کد به دست آمده است:

ابتدا ما با کدنویسی یک راه حل ساده و بی تکلف شروع کردیم. این به ما کمک کرد که دو مورد از تغییرات طراحی را متوجه شویم: سرعت و مصرف حافظه.

در مرحله بعد، ما یک طرح تسمه نقاله را امتحان کردیم. این طراحی سرعت و مصرف حافظه را بهبود بخشید، اما هنوز به اندازه کافی برای کاربردهایی با کارآیی عالی مناسب نبود. همچنین این طراحی انعطاف پذیر نبود و تطبیق پذیری کد برای مدیریت دیگر فواصل زمانی نیازمند کار بسیاری بود.

طراحی نهایی ما مشکلات قبلی را با شکستن موارد در زیرمسئله‌ها حل کرد. در اینجا سه کلاس ایجاد شده به ترتیب پایین به بالا و زیرمسئله‌هایی که هر کدام حل کردند را داریم:

ConveyorQueue

حداکثر طول صفحه است که می‌تواند جابجا شود و مجموع کل را نگهداری کند.

TrailingBucketCounter

را با توجه به مدت زمان سپری شده و نگهداری چندین فاصله زمانی تکی (آخرین) با دقتی خاص، جابجا می‌کند.

MinuteHourCounter

به سادگی شامل دو TrailingBucketCounters است. یکی برای شمارش دقیقه و دیگری برای شمارش ساعت.

پیشنهادهایی برای مطالعه



ما این کتاب را با بررسی هزاران نمونه کد نوشته‌یم تا بفهمیم که در عمل چه کاربردی دارد. اما همچنان از کتاب‌ها و مقالات زیادی که به ما در این امر کمک کردند نیز استفاده کردیم. اگر به یادگیری بیشتر علاقه دارید، در اینجا برخی از منابع آورده شده است که شاید دوست داشته باشید، آن‌ها را نیز مطالعه کنید. لیست‌های زیر به هیچ وجه کامل نیستند، اما برای شروع خوب هستند.

کتاب‌هایی برای نوشتن کد با کیفیت بالا

Code Complete: A Practical Handbook of Software Construction, 2nd edition, by Steve McConnell (Microsoft Press, 2004)

تحقیق بسیار خوبی است که تمام جنبه‌های ساخت نرم‌افزار از جمله کیفیت کد را پوشش داده است.

Refactoring: Improving the Design of Existing Code, by Martin Fowler et al. (Addison-Wesley Professional, 1999)

یک کتاب عالی که درباره فلسفه بهبود کد صحبت می‌کند و شامل فهرست مفصلی از بازسازی‌های مختلف، همراه با اقدامات لازم جهت انجام این تغییرات با کمترین مقدار از تقسیم کد است.

The Practice of Programming, by Brian Kernighan and Rob Pike (Addison-Wesley Professional, 1999)

این کتاب در مورد جنبه‌های مختلف برنامه‌نویسی از جمله اشکال‌زدایی، تست، قابلیت حمل و کارآیی بحث کرده و مثال‌های مختلفی از کدنویسی را بررسی می‌کند.

The Pragmatic Programmer: From Journeyman to Master, by Andrew Hunt and David Thomas (Addison-Wesley Professional, 1999)

مجموعه‌ای از اصول خوب برنامه‌نویسی و مهندسی که در بحث‌های کوتاه سازماندهی شده است.

Clean Code: A Handbook of Agile Software Craftsmanship, by Robert C. Martin (Prentice Hall, 2008)

این کتاب مشابه کتاب ما است (اما مخصوص جاوا) است. مباحث دیگری مانند مدیریت خطأ و همزمانی را نیز بررسی می‌کند.

کتاب‌هایی برای زبان‌های مختلف برنامه‌نویسی

JavaScript: The Good Parts, by Douglas Crockford (O'Reilly, 2008)

ما معتقدیم که این کتاب روح مشابهای مانند کتاب ما دارد، هر چند صریحاً درباره خوانایی نیست. این کتاب درباره استفاده از یک زیر مجموعه تمیز از زبان JavaScript است که خطای کمتر داشته و منطقی‌تر است.

Effective Java, 2nd edition, by Joshua Bloch (Prentice Hall, 2008)

یک کتاب خارق العاده در مورد آسانتر خواندن برنامه‌های جاوا و عاری از اشکالات. اگرچه این کتاب در مورد جاوا است، ولی بسیاری از اصول آن قابل تعمیم به همه زبان‌ها بوده و بسیار توصیه شده است.

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994)

این کتاب در مورد زبان مشترک الگوهای برای مهندسین نرم‌افزار جهت صحبت در مورد برنامه‌نویسی شئ‌گرا است. به عنوان یک فهرست عمومی و با الگوهای کاربردی، به برنامه‌نویسان کمک می‌کند که از خطاهایی که اغلب اتفاق می‌افتد جلوگیری کنند، آن هم زمانی که افراد برای اولین بار سعی می‌کنند یک مشکل ابهام آمیز را به تنها‌ی حل کنند.

Programming Pearls, 2nd edition, by Jon Bentley (Addison-Wesley Professional, 1999)

این کتاب نیز مجموعه‌ای از مقالات درباره مشکلات واقعی نرم افزارها است و بیش خوبی در مورد حل مشکلات دنیای واقعی در هر فصل ارائه می‌دهد.

High Performance Web Sites, by Steve Souders (O'Reilly, 2007)

اگرچه این کتاب در مورد برنامه‌نویسی نیست، اما قابل توجه است، زیرا تعدادی از روش‌های بهینه سازی یک وبسایت را بدون نوشتمن کد زیاد توضیح می‌دهد (با توجه به فصل ۱۳ - نوشتمن کد کمتر).

Joel on Software: And on Diverse and ..., by Joel Spolsky

برخی از بهترین مقالات مربوط به سایت <http://www.joelonsoftware.com> است. درباره بسیاری از جنبه‌های مهندسی نرم افزار مطلب می‌نویسد و در مورد موضوعات مختلفی روش‌نگری

می‌کند. حتماً دو مقاله «چیزهایی که هرگز نباید انجام دهید- قسمت اول^۱» و «تسنیت Joel: مرحله برای نوشتن کد بهتر^۲» را بخوانید.

کتاب‌هایی درباره نکات تاریخی^۳

Writing Solid Code, by Steve Maguire (Microsoft Press, 1993)

این کتاب متأسفانه کمی قدیمی شده است، اما مطمئناً مشاوره بسیار خوبی در مورد چگونگی بهتر کردن کد و کدنویسی بدون اشکال را بیان می‌کند که ما را تحت تاثیر قرار داد. اگر آن را بخوانید، متوجه می‌شوید که مطالب آن با پیشنهادات ما همپوشانی زیادی دارد.

Smalltalk Best Practice Patterns, by Kent Beck (Prentice Hall, 1996)

این کتاب اگرچه مثال‌های Smalltalk دارد ولی تعداد زیادی از اصول برنامه‌نویسی خوب را معرفی می‌کند.

The Elements of Programming Style, by Brian Kernighan and P.J. Plauger (Computing McGrawHill, 1978)

یکی از قدیمی‌ترین کتاب‌هایی که با موضوع «تمیزترین راه برای نوشتن چیزها» سر و کار دارد.

Literate Programming, by Donald E. Knuth (Center for the Study of Language and Information, 1992)

ما با دل و جان با دستور Knuth موافق هستیم، یعنی «به جای این که تصور کنیم وظیفه اصلی ما این است که به کامپیوتر آموزش دهیم که چه کاری را انجام دهد، بگذارید بیشتر تمرکز کنیم تا به انسان‌ها توضیح دهیم که از کامپیوتر چه کاری می‌خواهیم انجام دهد». اما هشدار داده شده است که: بخش عمدۀ کتاب مربوط به محیط برنامه‌نویسی وب برای مستندسازی است.

^۱ Things You Should Never Do, Part I

^۲ The Joel Test: 12 Steps to Better Code

^۳ Books of Historical Note