

اتکاپذیری نرم افزار

(Software Reliability)

فاز سوم: بررسی روش‌های ارتقاء اتکاپذیری در پروژه‌های کوچک

تهیه‌کنندگان: فاطمه ریاحی، نیما هزار، آیدین نصیری شرق

۵.....امروزی بر ادبیات موضوعی

- ۵..... 1.1 تعریف اتکاپذیری (Reliability) ۵
- ۵..... ۱.۲ تعریف شکست (Failure) ۵
- ۶..... 1.3 تعریف ترمیم یا Restore ۶
- ۶..... ۱.۴ تعریف دستگاه پوکایوک برای تولید نرم افزار ضد اشتباه (Mistake-Proof) ۶

۸..... ۲ مروری بر تعریف پروژه در فاز دوم

- ۸..... 2.1 اهمیت پرداختن به موضوع ۸
- ۸..... 2.2 دلیل انتخاب پروژه های «کوچک» ۸

۹..... 3 آشنایی با محیط

- ۹..... ۳.۱ تعریف پروژه ی کوچک ۹
- ۹..... ۳.۱.۱ ویژگی های ذاتی پروژه ی کوچک ۹
- ۹..... ۳.۱.۲ کاربردهای پروژه ی کوچک ۹
- ۹..... ۳.۱.۳ ویژگی های معلول در پروژه ی کوچک ۹
- ۱۱..... ۳.۲ تطبیق محیط با تعاریف علمی ۱۱
- ۱۱..... ۳.۲.۱ پیش گیری و تشخیص در پروژه های کوچک ۱۱
- ۱۲..... ۳.۲.۲ تصحیح کننده ها ۱۲
- ۱۳..... ۳.۲.۳ انطباق دستگاه های پوکایوک بر پروژه های کوچک ۱۳

۱۵..... ۴ تکنیک های پیش نهادی برای ارتقاء اتکاپذیری پروژه های کوچک

- ۱۵..... ۴.۱ آشنایی با محیط ۱۵
- ۱۵..... ۴.۲ ویژگی های تکنیک مناسب برای ارتقاء اتکاپذیری ۱۵
- ۱۶..... ۴.۳ معرفی تکنیک ها ۱۶
- ۱۶..... ۴.۳.۱ گروه صفر، تکنیک های مستقل از کد ۱۶
- ۱۶..... ۴.۳.۱.۱ تکنیک صفر، انتخاب زبان برنامه سازی مناسب ۱۶

تکنیک اول، انتخاب کامپایر مناسب و استفاده درست از آن.....	۴.۳.۱.۲	۱۷
تکنیک دوم، انتخاب ویرایشگر متنی مناسب و استفاده راحت از آن.....	۴.۳.۱.۳	۱۷
تکنیک سوم، انتخاب دیباگر مناسب.....	۴.۳.۱.۴	۱۹
تکنیک چهارم، انتخاب زمان و مکان مناسب!.....	۴.۳.۱.۵	۱۹
تکنیک پنجم، درک درست صورت مسئله.....	۴.۳.۱.۶	۲۰
۴.۳.۲ گروه اول، تکنیک‌های عمومی		
تکنیک اول، جداسازی خروجی‌ها.....	۴.۳.۲.۱	۲۰
تکنیک دوم، چاپ سریع متغیرها.....	۴.۳.۲.۲	۲۲
تکنیک سوم، کامنت (و آن کامنت) کردن سریع یک تکه از کد.....	۴.۳.۲.۳	۲۲
تکنیک چهارم، پایان پروژه چند ساعت پس از پایان پروژه است!.....	۴.۳.۲.۴	۲۲
۴.۳.۳ گروه دوم، پیش‌گیری از خطاهای اعمال شده‌ی خارجی		
تکنیک اول، بررسی کردن تعاملات با سیستم.....	۴.۳.۳.۱	۲۳
تکنیک دوم، تست‌ها، بررسی و تست‌های بی‌جواب.....	۴.۳.۳.۲	۲۳
تکنیک سوم، هیچ‌چیز ۱۰۰٪ درست نیست!.....	۴.۳.۳.۳	۲۳
۴.۳.۴ گروه سوم، پیش‌گیری و تشخیص خطاهای درون کد		
تکنیک اول، استفاده از ثابت‌ها (const).....	۴.۳.۴.۱	۲۴
تکنیک دوم، همه‌ی ثابت‌ها int نیستند.....	۴.۳.۴.۲	۲۴
تکنیک سوم، تعریف مجدد تایپ‌های ساده (نظیر int) به نام‌های مفهوم‌دار.....	۴.۳.۴.۳	۲۵
تکنیک چهارم، تعریف متغیرها، هر چه نزدیک‌تر به محل استفاده.....	۴.۳.۴.۴	۲۵
تکنیک پنجم، پرهیز از ارسال با مرجع (reference) به پارامترهای تابع، مگر با کمک const.....	۴.۳.۴.۵	۲۵
تکنیک ششم، حداکثر استفاده از توابع و ساختارهای خود زبان.....	۴.۳.۴.۶	۲۶
تکنیک هفتم، هر حلقه‌ای پایان می‌یابد.....	۴.۳.۴.۷	۲۶
تکنیک هشتم، پرهیز از گزینه‌های منوی Edit نظیر copy و paste و نیز replace.....	۴.۳.۴.۸	۲۷
تکنیک نهم، پرهیز از خروج با exit().....	۴.۳.۴.۹	۲۷
تکنیک دهم، نام‌گذاری سازگار (consistent) و با معنی.....	۴.۳.۴.۱۰	۲۷

۴.۳.۴.۱۱ تکنیک یازدهم، مقداردهی مقادیر بی‌ربط به‌عنوان مقادیر اولیه ۲۸

۴.۳.۴.۱۲ تکنیک دوازدهم، قانون ساده‌ی اولویت عملگرها ۲۸

۴.۳.۴.۱۳ تکنیک سیزدهم، صرفه‌جویی نکردن بی‌مورد ۲۹

۴.۳.۵ گروه چهارم و آخر، تکنیک‌های نهایی ۳۰

۴.۳.۵.۱ تکنیک اول، برداشت صحیح از زمان اجرا ۳۰

۴.۳.۵.۲ تکنیک دوم، کثیف رسیدن به‌تر از هرگز نرسیدن است! ۳۰

۴.۳.۵.۳ تکنیک سوم، تنبل نباشید! ۳۰

۵ جمع‌بندی ۳۱

۶ پیش‌نهاداتی برای کارهای آینده ۳۲

۷ منابع ۳۳

۱ مروری بر ادبیات موضوعی

۱.۱ تعریف اتکاپذیری (Reliability)

چنانچه در بخش ۳.۴ از کتاب درس [1] می‌خوانیم، اتکاپذیری یا Reliability از فاکتورهای سنجش کارایی نرم‌افزار که برخلاف سایر فاکتورها (نظیر خوانایی^۱، کیفیت، پیچیدگی و ...) به‌صورت مستقیم و از روی داده‌های آماری و تاریخی به‌دست می‌آید. در فصل ۸.۸ کتاب که مشخصاً به اتکاپذیری تخصیص یافته، یک تعریف آماری برای اتکاپذیری از [2] بدین‌گونه ارائه می‌شود:

احتمال عاری از بدون شکست بودن (Failure Free) یک برنامه‌ی کامپیوتری در یک محیط مشخص و در یک زمان مشخص را میزان اتکاپذیری آن برنامه می‌نامند.

برای مثال هنگامی که میزان اتکاپذیری یک نرم‌افزار ۹۶٪ برای ۸ ساعت کار متوالی تعریف می‌شود بدین معنی است که اگر این برنامه ۱۰۰ مرتبه (و هر مرتبه برای ۸ ساعت) اجرا شود، در ۹۶ مرتبه کار خود را بدون شکست و با موفقیت به‌پایان می‌رساند.

۱.۲ تعریف شکست (Failure)

در ادامه‌ی تعریف اتکاپذیری، کلمه‌ی شکست یا Failure می‌بایست تعریف شود. در منبع اصلی (کتاب درس) اشاره‌شده در تمام تعریف‌های شکست در محافل کیفیت و اتکاپذیری نرم‌افزار تعریف زیر به چشم می‌خورد

شکست عبارتست از عدم همخوانی (nonconformance) نرم‌افزار با خواسته‌ها (requirement)ی آن.

البته باید اشاره کرد که شکست‌ها غالباً ناشی از خطاها (Errors)ی نرم‌افزار - چه در کدنویسی، چه در طراحی - هستند و ممکن‌ست بسیار فاجعه‌بار یا بسیار کوچک باشند. همچنین باید به‌خاطر داشت که هر خطایی لزوماً به شکست منجر نمی‌شود و چه بسا نرم‌افزارهای در حال استفاده‌ای که دارای خطاهای کشف نشده‌اند!

در سوی دیگر، در یک رویکرد ریشه‌ای به دلایل پیدایش شکست در نرم‌افزار می‌تواند به‌این نتیجه رسید که اکثر شکست‌های نرم‌افزاری ناشی از نقایص طراحی و پیاده‌سازی هستند و نه دلایلی نظیر فرسودگی یا عوامل فیزیکی (نظیر آن‌چه در سخت‌افزار مطرح می‌شود). از این‌رو افزایش اتکاپذیری نرم‌افزار عملاً بسته به ریشه‌یابی و ترمیم شکست‌های احتمالی آن دارد.

¹ Readability

۱.۳ تعریف ترمیم یا Restore

در عمل، یکی از معیارهای سنجش بزرگی (شدت) یک شکست را می‌توان پیچیدگی و زمان‌بری تصحیح آن دانست. گاه یک شکست کوچک را می‌توان در چند ثانیه رفع (ترمیم) کرد (مخصوصاً اگر در مراحي اولیه‌ی طراحی و پیش از انتشار پروژه باشد) و گاه برطرف نمودن یک شکست می‌تواند به هفته‌ها یا ماه‌ها زمان احتیاج داشته و حتی کل پروژه را نابود کند.

اهمیت زمان ترمیم هنگامی آشکار می‌شود که می‌بینیم کتاب مرجع درس میزان ساده از اتکاپذیری به نام $MTBF^2$ یا «زمان میانگین بین شکست» را چنین تعریف می‌کند:

$$MTBF = MTTF + MTTR$$

یا

زمان میانگین ترمیم + زمان میانگین شکست = زمان میانگین بین شکست

که فی‌الواقع بزرگی شکست را با بزرگی زمان لازم برای ترمیم آن هم‌سان دانسته اند. دکتر پرسمن در اهمیت $MTBF$ اشاره می‌کند که این فاکتور به مراتب مفیدتر از تعداد نقایص به‌ازای هر KLOC یا تعداد نقایص به‌ازای هر FP است.

علاوه بر این، فاکتور دیگری که همگام با اتکاپذیری نرم‌افزار تعریف شده، قابلیت دسترسی یا $Availability$ است که مشابهاً با استفاده از $MTTR$ و $MTTF$ به‌دست می‌آید:

$$Availability = [MTTF / (MTTF + MTTR)] \times 100\%$$

این فاکتور به وضوح اهمیت $MTTR$ را نشان می‌دهد.

۱.۴ تعریف دستگاه پوکایوک برای تولید نرم‌افزار ضد اشتباه (Mistake-Proof)

با مقدمه‌ی بالا، به این نتیجه می‌رسیم که در راستای بهبود و ارتقاء اتکاپذیری یک نرم‌افزار دو راهکار وجود دارد:

۱. پیش‌گیری از مشکلات کیفیتی بالقوه قبل از رخداد آن‌ها (اصطلاحاً پیش‌گیری یا Prevention)

۲. تشخیص سریع مشکلات کیفیتی در صورت وارد شدن آن‌ها (اصطلاحاً تشخیص یا Detection)

دکتر پرسمن در کتاب مهندسی نرم‌افزار خود (مرجع اول) اشاره می‌کند که یک مهندس صنایع ژاپنی به‌نام شینگو شینگو [3] که در شرکت تویوتا کار می‌کرده، تکنیکی برای تضمین کیفیت ابداع کرده که عیناً بر پایه‌ی دو راهکار بالا یعنی «پیش‌گیری» و «تشخیص» است. این تکنیک به‌طور ضمنی و گاه غیرمستقیم در بسیاری از جزئیات زندگی روزمره‌ی ما وجود دارد؛ برای مثال در اتومبیل‌های جدید اگر دنده‌ی اتومبیل خلاص نباشد، استارت کار نمی‌کند (پیش‌گیری) و اگر کمر بند ایمنی بسته نباشد، بوق یا چراغ هشدار به‌صدا در می‌آید (تشخیص). حتی در محصولات صنعتی جدید، خصوصاً در کشور ما، این دو تکنیک افزایش

² MTBF: Mean Time Between Failure

اتکاپذیری به عنوان عاملی در تعریف [تجاری، تبلیغاتی] هوشمندی یا **Intelligence** محصولات به شمار می‌رود؛ برای مثال اجاق گازی که اجازه‌ی پیچاندن دستگیره‌ی پخش گاز را پیش از روشن کردن فندک نمی‌دهد (عیناً پیش‌گیری) یک محصول هوشمند به شمار می‌رود و اگر به‌هنگام سرریز غذا روی شعله، هشدار (چراغ، بوق یا حتی قطع گاز) بدهد (تشخیص) بر هوشمندی آن محصول افزوده می‌شود.

با این وصف، یک دستگاه ضد اشتباه (که با لفظ ژاپنی آن پوکایوک^۳ خوانده می‌شود) باید ویژگی‌های زیر را داشته باشد:

- **آسان و ارزان باشد:** چرا که اگر بیش از حد پیچیده باشد، خود نیاز به یک پوکایوک خواهد داشت(!) و اگر گرانقیمت باشد، توجیه اقتصادی پیدا نمی‌کند.
- **بخشی از فرآیند باشد:** در محصول به صورت مختلط و منسجم (**integrated**) قرار بگیرد و به صورت یک ساختار خارجی (نظیر ابزارهای **test**) قرار نگیرد.
- **در نزدیکی وظیفه‌ای که در آن امکان رخداد اشتباه وجود دارد قرار بگیرد:** با این ویژگی امکان بازخورد سریع و تصحیح خطا فراهم می‌آید.

³ Poka - yoke

۲ مروری بر تعریف پروژه در فاز دوم

پس از بررسی حوزه‌ی اتکاپذیری نرم‌افزار^۴، با مشورت با اعضای گروه تصمیم بر این گرفته شد که ادامه‌ی کار حول «بررسی روش‌های ارتقاء اتکاپذیری در پروژه‌های کوچک» باشد. دلایل اصلی انتخاب این موضوع و اهمیت آن‌ها در زیر بررسی شده‌اند.

۲.۱ اهمیت پرداختن به موضوع:

طبق صحبت‌های دکتر میریان در جلسه‌ی گروه نرم‌افزار بهمن ماه سال گذشته، یکی از دلایل ضعف دانشجویان کارشناسی در مدیریت پروژه‌های بزرگ، عدم آشنایی آن‌ها با این مهم از ابتدای کار است. ایشان معتقد بودند که دانشجوی کارشناسی وقتی از ابتدا (در درس مبانی برنامه‌سازی) باید پروژه‌ی بزرگی بنویسد و تنها ساختار زبان C++ به وی آموخته می‌شود، بالطبع به راه نادرستی عادت می‌کند. ما قصد داریم با بررسی روش‌های ارتقاء اتکاپذیری در پروژه‌های ابتدایی، علاوه بر امکان‌سنجی گزینه‌های موجود، راه‌کارهای کاربردی این مهم را نیز پیدا کنیم که می‌تواند در آموزش نیز مؤثر باشد.

۲.۲ دلیل انتخاب پروژه‌های «کوچک»:

چنانچه در بخش ۱.۹ فاز یک پروژه اشاره شد، بررسی‌کننده‌ها و خودتصحیح‌کننده‌ها می‌توانند به‌صورت جزئی روی بخش‌های مختلف یک پروژه به‌صورت جدا اعمال شوند. در واقع این انتزاع به ما امکان می‌دهد تا برای هر یک از بخش‌های مختلف یک پروژه‌ی بزرگ (نظیر I/O، UI، تعاملات DB و ...) به‌طور جدا یک بررسی‌کننده داشته باشیم تا در نهایت جمیع آن‌ها مانند یک بررسی‌کننده‌ی بزرگ عمل کنند.

با این تفسیر، در صورتی که ما بتوانیم بررسی‌کننده‌هایی کارا بر روی پروژه‌های کوچک (زیر ۱۰۰۰ خط کد) ارائه دهیم، در واقع اجزاء تشکیل‌دهنده‌ی یک کد بزرگ را نیز بهبود بخشیده‌ایم.

یک دلیل دیگر برای انتخاب پروژه‌های کوچک، تسلط تهیه‌کنندگان بر این سبک از پروژه‌هاست که بارها در درس‌های مختلف کارشناسی (نظیر «برنامه‌سازی مقدماتی»، «برنامه‌سازی پیش‌رفته»، «طراحی الگوریتم‌ها»، «شبیه‌سازی کامپیوتری»، «هوش مصنوعی»، «سیستم‌های عامل» و ...) مخصوصاً در دانشگاه صنعتی شریف ارائه شده و می‌شوند. با این تعریف، پروژه‌ی کوچک اصطلاحاً به پروژه‌هایی با مشارکت حداکثر ۳ نفر و غالباً کمتر از یک هفته اطلاق می‌شود.

لازم به تأکید است که از آنجا که (طبق گفته‌های فاز یک پروژه) اتکاپذیری معمولاً در اثر یک اشتباه کوچک نرم‌افزاری رخ می‌دهد، تأثیر سائز^۵ نرم‌افزار صرفاً در راستای پیچیدگی پیدا کردن اشتباه است که با اولاً در صورت ماژولار بودن سیستم این امر ساده‌تر می‌شود؛ ثانیاً از آنجا که «اتکاپذیری» یک روش پیش‌گیری و «خطا» اتفاق آتی است، این فاکتور عملاً بی‌معنی می‌شود.

^۴ Software Reliability

^۵ Scale

۳ آشنایی با محیط

۳.۱ تعریف پروژهی کوچک

چنان که گفته شد، هدف از انجام این پروژه «بررسی روش‌های ارتقاء آتکاپذیری در پروژه‌های کوچک» می‌باشد. برای این منظور، ابتدا «پروژهی کوچک» تعریف می‌شود.

۳.۱.۱ ویژگی‌های ذاتی پروژهی کوچک

در این مقاله، پروژهی کوچک به پروژه‌ای اطلاق می‌شود که ویژگی‌های زیر را ذاتاً داشته باشد:

- از نظر حجمی کوچک باشد - حدود ۱۰۰ الی ۳۰۰ خط و حداکثر ۵۰۰ خط باشد.
- از نظر تعداد افراد درگیر در کدنویسی کوچک باشد - حداکثر ۳ نفر در پروسه‌ی کدنویسی شرکت کنند.
- از نظر زمانی کوچک باشد - غالباً در یک وهله و حداکثر در ۳ روز نوشته می‌شود.

۳.۱.۲ کاربردهای پروژهی کوچک

چنین پروژه‌ای معمولاً در موارد زیر کاربرد دارد:

- مسابقات برنامه‌نویسی، تمرینات درسی، آکادمیک - که بیش‌تر در این مقاله مورد توجه قرار می‌گیرند
- خاص منظوره یا علمی - پیاده‌سازی یک الگوریتم جدید، محاسبه‌ی ریاضی یا یک ابزار ساده
- مستقل و قابل گسترش - که با پروژه‌های کوچک دیگر ترکیب‌شده و اجزاء یک پروژه‌ی بزرگ را می‌سازند

تأکید بیش‌تر این مقاله روی دو کاربرد اول و خصوصاً کاربرد آکادمیک است.

۳.۱.۳ ویژگی‌های معلول در پروژهی کوچک

ویژگی‌های این سبک پروژه‌ها، که بعضاً از تعریف و کاربرد آن ناشی می‌شود و آن‌را متمایز با سایر پروژه‌ها می‌کند، عبارتند از:

۱. غالباً بسیاری از فازهای مهندسی پروژه (اصطلاحاً گام‌های SDLC)، خصوصاً فاز طراحی را ندارند. دلیل

این امر در کاربردهای آکادمیک غالباً آشنا نبودن با این تکنیک‌ها و روش‌ها است زیرا در اکثر محیط‌های آموزشی (نظیر دانشگاه‌های کشور ما)، درس‌های مهندسی نرم‌افزار (نظیر «طراحی و تحلیل سیستم‌ها» و «مهندسی نرم‌افزار») در سال‌های آخر تحصیل یاد داده می‌شوند در حالی که کدنویسی خام در ترم اول آموزش داده شده و در اکثر ترم‌ها طی تمرینات و پروژه‌های درسی مورد کاربرد قرار می‌گیرند. در کاربردهای خاص منظوره نیز به‌صرفه نبودن اقتصادی و یا اصطلاحاً «یک‌بار مصرف بودن» کد مانع اعمال فاز «طراحی» و حتی «تحلیل» می‌شود و نهایتاً در کدهای مستقل، غالباً انتزاع و افراز موجود امکان/نیاز طراحی را سلب می‌کند.

۲. خطاهای انسانی در آن بسیار دخیل است. چنانچه در مقاله‌ی [4] اشاره شده است، خطاهای انسانی در پروژه‌های کوچک غالباً از چهار منبع ناشی می‌شوند:

۰ **اتکای زیاد بر حافظه‌ی کدنویس** - به دلیل نبودن مستند و احساس بی‌نیازی به کوچک‌ترین رفتار در راستای مستندسازی

۰ **تصمیم‌گیری‌های لحظه‌ای** - مثلاً در نحوه‌ی پیاده‌سازی یک تابع کوچک یا نام‌گذاری متغیرهای عمومی

۰ **تهییج برای استفاده از حداکثر توانایی‌ها** - که غالباً در کدنویسان جوان رخ داده و آن‌ها را به استفاده

از تکنیک‌های نامطمئن (مثلاً استفاده از تابع دیگری که به تازگی یادگرفته شده) سوق می‌دهد. ترقیب این دسته از کدنویسان هنگامی افزایش می‌یابد که فکر می‌کنند کد کوچک بستر مناسبی برای یادگیری استفاده از ابزارهای جدید است، حال آن‌که به دلیل پایین بودن کیفیت تست (و حتی، بعضاً در کاربردهای آکادمیک، عدم امکان از آگاهی شکست و علت‌یابی) ممکن است باعث شود تا اطمینان شخصی کد نویس روی یکی تکنیک غلط (در کاربرد و نحوه استفاده) بالا رفته و او این خطا را بارها انجام بدهد.

۰ **محدودیت‌های ارتباطی** - عدم امکان/نیاز به مشاوره با کارفرما (ناظر، supervisor) که معمولاً ناشی از مدت کم انجام کار است می‌تواند منبع بسیاری از مشکلات آتی باشد. این مهم در اکثر منابع به **clear communication** تعبیر می‌شود.

۳. غالباً از قابلیت خوانایی (**Readability**) پایینی برخوردارند. ویژگی‌های تعریفی دوم و سوم (کم بودن تعداد افراد درگیر و در یک بازه‌ی زمانی محدود) دلیل اصلی این مهم است. کدنویس (یا هر یک از چند کدنویس محدود) تنها خوانندگان کدی هستند که تنها نتیجه‌ی آن، و برای مدتی محدود (گرفتن نمره در کاربردهای آکادمیک و نیل به جواب در کارهای خاص منظوره) اهمیت دارد. در این شرایط، کدنویس غالباً سرعت نوشتن را فدای خوانایی کرده و از انجام کارهایی نظیر گذاشتن توضیحات (**comment**) در کد و اصول پایه‌ی کد استاندارد خودداری می‌کند.

۴. در صورت بروز خطا، هزینه‌ی بسیار زیادی را دربرنخواهند داشت. دلیل این امر محدود بودن استفاده‌کنندگان و زمان استفاده است. متأسفانه به این دلیل است که برنامه‌نویسان باتجربه نیز انجام اعمال بسیار ساده در راستای ارتقاء اتکاپذیری (که در ادامه ذکر می‌شود) را نادیده می‌گیرند.

۵. زمان، در مقیاس بسیار کم (کم‌تر از یک هفته) اهمیت بسیار بالایی دارد. معمولاً در این پروژه‌ها، به دلیل موقعیت و شرایط کدنویس و هدف پروژه (که کم‌تر درگیر مسائل اقتصادی می‌شود)، زمان بر کیفیت ارجح است. جدای از کم‌تجربی، شاید همین فاکتور «زمان» باشد که اکثر برنامه‌نویسان را از تست (به صورت علمی) باز می‌دارد.

۶. شروع کار غالباً به صورت **trigger action** است. برخلاف پروژه‌های بزرگ که پیش از شروع به کزدن (**implementation**) زمان زیادی صرف آشنایی، تحلیل و ... می‌شود، در این سبک از پروژه‌ها اولین تماس با پروژه، شروع کار کزدن است و هر تصمیم‌گیری صرفاً در زمان نیاز/وقوع صورت می‌پذیرد. اصطلاحاً سیاق انجام این پروژه‌ها را می‌توان همواره بالا به پایین (**top-down**) دانست که فارغ از هرگونه آینده‌نگری است.

۷. ممکن است شکست این پروژه‌ها ریشه‌یابی نشود و یا به انجام دوباره‌ی آن بیانجامد. برای مثال در کاربردهای آکادمیک، کم‌تر پیش می‌آید که دانشجویی پس از آگاهی از نگرفتن نمره (در برخی تست‌ها) اقدام به ریشه‌یابی شکست بکند. به عبارت دیگر، پس از انتشار اولیه‌ی پروژه (ارسال آن به مصحح تمرینات)، امکان ترمیم پروژه فراهم نیست. نیز در سایر کاربردها (خاص منظوره یا گسترش‌پذیر)، راهکار مورد استفاده (در صورتی که زمان تشخیص و ترمیم خطا از حدی بیش‌تر بشود) غالباً انجام دوباره‌ی کار است. هم‌چنین شکست در این پروژه‌ها غالباً به صورت صفر و یک (شکست یا موفقیت) معنا می‌شود و شکست نسبی چندان معنا ندارد.

۳.۲ تطبیق محیط با تعاریف علمی

پس از تعریف محیط کاری و ویژگی‌های پروژه‌ی کوچک، تعاریف علمی گفته شده در بخش اول را بر این محیط منطبق می‌کنیم.

۳.۲.۱ پیش‌گیری و تشخیص در پروژه‌های کوچک

چنان‌چه در تعاریف علمی «اتکاپذیری» گفته شد، «پیش‌گیری» یا Prevention و «تشخیص» یا Detection دو گروه اصلی اقدامات در زمینه‌ی افزایش ارتقاء اتکاپذیری را تشکیل می‌دهند. طبق تعریف، «پیش‌گیری» عملیاتی است که از امکان رخ دادن خطا جلوگیری می‌کند و «تشخیص» عملیاتی است که پس از رخ دادن خطا، محل و دلیل رخ دادن آن را اطلاع می‌دهد تا فرآیند ترمیم را تسریع کند.

به‌طور خاص، در پروژه‌های کوچک به‌دلیل خاصیت «عدم وجود مراحل شکست» غالباً «تشخیص» و «پیش‌گیری» در عمل اشتراک بسیار زیادی دارند و تمایز این دو عملاً بسیار ظریف است. مثال زیر از «تشخیص» را در نظر بگیرید

```
if (a == 0) {  
    print("Predicted Error: Can not divide by zero in (b = c / a)");  
    exit(1); // Exit before failure  
}  
b = c / a;
```

مشاهده می‌شود که در این کد به محض شناسایی خطا، از بروز آن خودداری به‌عمل آمده و خروج از سیستم تحت کنترل (اصطلاحاً نرمال) خواهد بود.

اما در تکنیک‌های تشخیص، صرفاً وقوع خطا اعلان می‌شود و برنامه مستقیم به سمت خطا می‌رود:

```
try {  
    if (a == 0)  
        print("Information: Divison is zero"); // just Detection, No Exit  
    b = c / a;  
} catch (DivisionByZeroException e) {  
    Print("Error Detected");  
}
```

که البته چنان که مشاهده می‌شود با تکنیک **Exception Handling** در این مثال، خطا «تشخیص» داده شده و الزاماً به شکست نیانجامیده است..

با این تفاسیر می‌توان وظایف پیش‌گیری و تشخیص را روی هم بدین‌صورت جمع‌بندی کرد (با کمک [5]):

- **شناسایی خطا و محل‌ها (دستورات خطاخیز)**، که تا حد امکان به‌محض وقوع خطا و پیش از آن‌که منجر به تخریب و شکست شود (پیش‌گیری)، با آن مقابله شود (پیش‌گیری و تشخیص)
- **اطمینان از شرایط و مقادیر در یک وهله‌ی خاص از برنامه**، که در عمل نتیجه‌ی اعمال وظیفه‌ی قبلی‌ست
- **شناسایی علّت خطا و تلاش برای آشکارسازی این علّت**، در صورت بروز خطا (تشخیص)
- **شناسایی رویه‌ی بروز خطا**، که در عمل خطوط طی شده تا محل خطا و (برای مثال) مقادیر متغیرها در هنگام خطا را اعلام می‌کند (تشخیص)

۳.۲.۲ تصحیح‌کننده‌ها

علاوه بر این موارد، می‌توان (مطابق تحقیقات فاز یک پروژه) دسته‌ی دیگری از تکنیک‌های «افزایش اتکاپذیری» به‌نام «**تصحیح‌کننده‌ها**»^۶ را نیز به این مجموعه اضافه کرد که وظیفه‌ی اصلی آن‌ها چنین است:

- **تصحیح علّت خطاهای قابل تصحیح**، در صورت لزوم با اطلاع‌رسانی جانبی

یک مثال از چنین تکنیکی می‌تواند استقامت در برابر ورودی غلط باشد؛ چرا که با توجه به ویژگی‌های خاص پروژه‌های کوچک نظیر کم‌بودن استفاده‌کنندگان (ویژگی معلول شماره ۴) و محدودیت ارتباطی (ویژگی معلول شماره ۲، قسمت چهارم) ممکن‌ست ورودی برنامه پای‌بند به مشخصات گفته شده نباشد.

برای مثال برنامه‌ای را درنظر بگیرید که قرارست یک عبارت ریاضی را (به‌صورت یک رشته پشت‌سر هم از حروف و بدون فاصله) خوانده و مقدارش را محاسبه کند. دادن ورودی « $3 * 2 + 34$ » به این برنامه (که قبل از $*$ آن فاصله خالی وجود دارد) ممکن‌ست باعث شکست برنامه شود، حال آن‌که عملیاتی نظیر جداسازی کلیه فاصله‌های خالی از رشته پیش از پردازش آن، یک عمل «تصحیح‌کننده» به‌شمار می‌رود.

البته باید توجه داشت که به‌دلیل ویژگی کمبود زمان (ویژگی معلول شماره ۵) پروژه‌های کوچک و نیز عدم تعهد (commitment) به حمایت بیش از اندازه از پروژه، تصحیح‌کننده‌های ورودی نادرست یا شرایط خاص در عمل کم‌تر مورد توجه

⁶ Corrector

قرار می‌گیرند. با این حال تصحیح‌کننده‌های خطاهای ناشی از خطاهای برنامه‌نویس را نباید از یاد برد. چنین تصحیح‌کننده‌هایی غالباً ماهیت مفهومی (Semantic) اجرای برنامه را بررسی کرده و در شرایط تشخیص خطا، برای جلوگیری از گسترش آن و شکست کل پروژه، تصمیمات لحظه‌ای اتخاذ می‌کنند که ممکن است لزوماً تصحیح کاملی نباشد.

در واقع می‌توان گفتن در تکنیک‌های «تشخیص» حدالمقدور سعی داریم رخ دادن خطا را آشکار کرده و گسترش دهیم تا راحت‌تر ریشه‌یابی شود، اما در تصحیح‌کننده‌ها تمام سعی ما بر این است که رخ دادن خطاها (ی برنامه‌نویس) حدالمقدور محدود بماند. برای مثال یک تصحیح‌کننده‌ی یک سیستم مدیریت کتابخانه می‌توان چنین باشد:

```
for each Member $i of MembersOfLibrary
    if ($BooksBorrowed[$i] < 0)
        $BooksBorrowed[$i] = 0;
```

که خطاهای ناشی از محاسبه‌ی اشتباه تعداد کتب امانت گرفته توسط یک کاربر را حداقل امکان تسطیح می‌سازد. البته باید توجه داشت که مقدار جایگزین صفر باید با دقت انتخاب شود و جدای از این، این خطا باید حتماً به مدیر برنامه اطلاع داده شود:

```
for each Member $i of MembersOfLibrary
    if ($BooksBorrowed[$i] < 0) {
        Print("member "+$i+" has invalid no. of books, I just corrected
it by myself.");
        $BooksBorrowed[$i] = 0;
    }
```

نهایتاً باید اشاره کرد که اولاً نوشتن یک تصحیح‌کننده‌ی مناسب (که اشتباهاً مقادیر درست را تحت تأثیر قرار ندهند و هم‌چنین مقدار/عمل جایگزین را هوشمندانه انجام بدهد) دشوار و حتی بعضاً دشوارتر از خود برنامه‌است. ثانیاً باید به‌خاطر داشت که اکثر برنامه‌نویسان حرفه‌ای معتقدند که

تعداد تشخیص‌دهنده‌ها پیش از تحویل و انتشار محصول باید تا حد ممکن و معقول زیاد باشد و در هنگام تحویل، تمام آن‌ها حذف شده و یا با تصحیح‌کننده‌ها جایگزین شوند.

۳.۲.۳ انطباق دستگاه‌های پوکایوک بر پروژه‌های کوچک

دیدیم که یک دستگاه ضد اشتباه (پوکایوک) ۳ ویژگی «ارزان و ساده بودن»، «مجتمع و منسجم شدن در داخل محصول» و «جای‌گیری مناسب» را باید داشته باشد. با مروری بر تعریف پروژه‌ی کوچک به این نتیجه می‌رسیم که:

تکنیک ضد/اشتباه (پوکایوک) در پروژه‌های کوچک باید ساده و مختصر بوده و در بدنه‌ی اصلی‌ی کد قرار بگیرند.

به طور دقیق تر می توان اشاره کرد که در راهکار پوکایوک، برای هر عملی که امکان اشتباه (ولو با احتمال بسیار کم) وجود دارد، پوکایوک مربوط به آن باید در همان قطعه از کد و در همان مکان قرار بگیرد. برای مثال یک اگر فراخوانی تابع RiskyFunc() در شبه کد زیر احتمال شکست وجود دارد، یک تکنیک پوکایوک می تواند مشابه آن چه نوشته شده است، باشد:

```
Print ("Calling RiskyFunc()"); // not necessarily to program's output!  
RiskyFunc();  
Print("RiskyFunc() Finished!");
```

در این مقاله، بر استفاده از این تکنیک تأکید شده و در ادامه کاربردهای «پیش گیری»، «تشخیص» و «تصحیح کننده گی» موارد مختلف در پروژه های کوچک معرفی می شوند.

۴ تکنیک‌های پیش‌نهادی برای ارتقاء اتکاپذیری پروژه‌های کوچک

۴.۱ آشنایی با محیط

آن‌چا تا بدین جا گفته شد، معرفی‌ای بود از اتکاپذیری و روش‌های رایج و مناسب آن در پروژه‌های کوچک. در این بخش قصد داریم به‌صورت خاص به معرفی تکنیک‌های رایج و مناسب (با نگاه کاملاً عملی) در این زمینه بپردازیم.

در بیان تکنیک‌های زیر، از زبان C++ که یکی از رایج‌ترین، قدرت‌مندترین و در عین‌حای خطاپذیرترین زبان‌های برنامه‌سازی است بهره گرفته‌ایم. زیر این زبان در اکثر دانشگاه‌ها (خصوصاً دانشگاه صنعتی شریف) به‌عنوان زبان پایه‌ی آموزش برنامه‌سازی مورد استفاده قرار می‌گیرد. البته زبان Java (که رقیب C++ در آموزش است)، نیز می‌توانست استفاده شود اما به‌دلیل سیاست‌های داخلی و ذاتی آن‌زبان برای اجتناب از خطا، خلوص عمل عملاً از دست می‌رفت و نتیجتاً C++ ترجیح داده شد.

۴.۲ ویژگی‌های تکنیک مناسب برای ارتقاء اتکاپذیری

تکنیک‌هایی که در ذیل معرفی می‌شوند، عمدتاً با توجه به ویژگی‌هایی انتخاب و ابداع شده‌اند که اهم آن‌ها را می‌توان چنین

برشمرد:

۱. **سادگی:** این ویژگی به‌طور مفصل در تعریف پوکایوک گفته شد. در عمل سعی می‌شود که تکنیک‌های مناسب حدالمقدور بیش از حدود ۵ خط نداشته باشند.
۲. **قابلیت قرارگیری در درون کد:** این ویژگی، عملاً ویژگی سوم پوکایوک است که در محیط ما، سعی شده که جاگذاری پوکایوک در کد لطمه‌ای به کارایی (Performance) و خوانایی (Readability) کد نزند.
۳. **سهولت یادگیری:** سعی شده حدالمقدور در نکات و تکنیک‌هایی که در ذیل مطرح می‌شوند از عملیات تخصصی و پیچیده‌ی زبان استفاده نشود و در صورت استفاده، به‌دقت توضیح داده شوند.
۴. **اختصار، کلی‌نگری و نگرشی بر خلاقیت:** در هر تکنیک، سعی شده است که پایداری به جامعیت موضوع در اولویت قرار بگیرد و از پرداختن به جزئیات خاص خودداری شود. بدیهی‌ست که خواننده می‌تواند یک تکنیک را خود به چندین تکنیک کوچک‌تر تقسیم کند و با خلاقیت خود، رهیافت مناسبی را اتخاذ کند.

۴.۳ معرفی تکنیک‌ها

۴.۳.۱ گروه صفر، تکنیک‌های مستقل از کد

۴.۳.۱.۱ تکنیک صفر، انتخاب زبان برنامه‌سازی مناسب

برای نوشتن یک پروژه ولو کوچک، حدالمقدور از زبان برنامه‌سازی ناآشنا و نامطمئن استفاده نکنید. استفاده از زبانی که تسلط کافی بر آن ندارید (حتی با در دست داشتن Manual مناسب) ممکن‌ست باعث گیج‌شدن شما و استفاده نامناسب و نادرست از امکانات زبان بشود. دقت کنید که بسیاری از نکاتی که به‌نظر یک برنامه‌نویس ممکن‌ست بدیهی باشد، از نظر طراحان یک زبان برنامه‌نویسی ممکن‌ست به دلایل خاص (و قطعاً علمی و سودمند) به‌صورت غیر رایج ارائه شوند.

برای مثال کد زیر در زبان C++ را در نظر بگیرید:

```
string s = "abcdef";  
cout << s.substr(2, 3) << endl; // prints "cde"
```

دو پارامتر ارسال شده به تابع زیر رشته (substr) روی رشته‌ی s، به‌ترتیب ایندکس شروع و طول زیردنباله‌ی مورد درخواست هستند که علی‌رغم تعریف ریاضی $s[2..3]$ ، زیر رشته‌ای با شروع از خانه ۲ و به‌طول ۳ (نه تا خانه ۳) از s را برمی‌گرداند.

یک برنامه‌نویس تازه‌کار که مسلط به زبان C++ نباشد، ممکن در چنان مواقعی دچار شک شده و معنای اشتباه را برداشت کند.

مثال دشوارتر از این مورد کد زیر است که برای یکی از برنامه‌نویسان ماهر دانشکده ایجاد مشکل کرد!

```
string s = "abcdef";  
for (int i = -1; i < s.size(); i++)  
    if (i >= 0)  
        cout << s[i] << endl; // never reaches here!  
    else  
        cout << "i is negative : " << i << endl; // neither here!
```

علی‌رغم شهود عینی که می‌بایست تمام کاراکترهای رشته‌ی s در خروجی چاپ شوند (و یا خط i is negative با اجرای این کد خواهیم دید که هیچ‌چیز در خروجی چاپ نمی‌شود! در صورتی که کد با پارامتر Wall- (که در تکنیک بعدی معرفی می‌شود)، کامپایل شود هشدار می‌بندی بر « warning: comparison between signed and unsigned integer expressions » دریافت می‌کنیم که نشان می‌دهد در شرط $i < s.size()$ سمت چپ از نوع (signed) int و سمت راست از نوع unsigned است. شاید کم‌تر برنامه‌نویس (حرفه‌ای) توجه کند که در زبان C اولویت unsigned بیش‌تر از int است و در مواقع مقایسه‌ی این دو نوع، مقدار int به unsigned (و نه unsigned به int) تبدیل شده و سپس مقایسه انجام می‌شود!

با این دو مثال، اهمیت انتخاب زبان برنامه‌نویسی در مواقعی که امکان انتخاب وجود دارد به‌وضوح آشکار می‌شود. در چنین مواقعی، تسلط بر یک زبان اهمیت بسیار بالایی دارد و حتی در مواقعی که امکانات آماده‌ی زبان ثانوی و سوسه‌انگیز است (نظیر کتابخانه‌های آماده، یا ابزار قابل دالود) می‌بایست مورد ارزش‌یابی درست قرار بگیرد.

۴.۳.۱.۲ تکنیک اول، انتخاب کامپایلر مناسب و استفاده درست از آن

کامپایلر (بعد از نویسنده)، اولین کسی (برنامه‌ای) است که کد را می‌خواند و می‌تواند خطاهای محتمل کد را شناسایی و هشدار بدهد. با رشد کامپایلرها و قدرت‌مندتر شدن آنها، اکثر کامپایلرهای پیش‌رفته نظیر `g++` (که در این مقاله کامپایلر اصلی زبان `C++` فرض می‌شود) امکاناتی در راستای تشخیص و پیش‌گیری از خطا دارند که مطلوب‌ست یک برنامه‌نویس جوان حتماً از آنها آگاهی پیدا کند.

مهم‌ترین این امکانات در زبان `C++` و کامپایلر `g++` پارامتر خط فرمان `-Wall` (مخفف `all warnings`) است که به‌صورت زیر در هنگام کامپایل به کد داده می‌شود:

```
C:\>g++ a.cpp -Wall
```

این امکان تعداد بسیار زیادی از خطاهای رایج کدها (نظیر قرار دادن یک `=` به جای `==` در شرط `if` یا `initial` نکردن متغیرها یا بی‌استفاده ماندن متغیرها را یادآوری می‌کند.

جدای از این مسئله، استفاده از یک کامپایلر نامطمئن (نظیر کامپایلرهای نامعروف یا نسخه‌های `beta`ی یک کامپایلر) ممکن‌ست برنامه‌نویس معصوم را دچار خطاهای ناخواسته بکند! نمونه‌ای از مورد، خطای یکی از نسخه‌های `prerelease` کامپایلر `g++` در تخصیص فضای `heap` بوده است که باعث شد یکی از نویسندگان این مقاله، ساعت‌ها در گیر کدش باشد.

۴.۳.۱.۳ تکنیک دوم، انتخاب ویرایشگر متنی مناسب و استفاده راحت از آن

انتخاب ویرایشگر متنی مناسب از نکات حائز اهمیت در پیش‌گیری از خطا در برنامه‌سازی است. در واقع یک ویرایشگر متنی، می‌تواند دستیار برنامه‌نویس بوده و وی را از بسیار از خطاهای احتمالی آگاه سازد.

برای روشن‌تر شدن این موضوع، کد زیر را که از منبع [8] انتخاب شده است، در نظر بگیرید:

```
1 #include <iostream>
2 int main()
3 {
4     int base = 0;    /* Base of the triangle */
5     int height = 0; /* Height of the triangle */
6
7     base = 5;        /* Set the base of the triangle
8     height = 2;      /* Set the height */
```

```

9
10 // Area of the triangle
11 int area = (base * height) / 2;
12
13 std::cout << "The area is " << area << std::endl;
14 return (0);
15 }

```

شاید برای شما هم جالب باشد که ببینید خروجی این برنامه صفر (و نه ۵) است! دلیل این امر را می‌توانید با کپی کردن این کد در یک ادیتور مناسب (که قابلیت **Syntax Highlighting** دارد مشاهده کنید (در صورتی که فرصت انجام این کار را ندارید، به انتهای خط شماره ۷ با دقت توجه کنید)! یک ادیتور هوشمند می‌توان با قابلیت **Syntax Highlighting** (که شاید از نظر برنامه‌نویسان جوان یک ابزار تجملاتی به‌شمار برود)، عملاً برنامه‌نویس را از بسیار از خطاها آگاه سازد.

به‌عنوان مثالی دیگر، این کد را در نظر بگیرید

```

cin >> x;
if (x % 2 == 0)
    if (x % 3 == 0)
        cout << "X is a multiplier of 6" << endl;
else
    cout << "blah! X is odd!" << endl;

```

علی‌رغم انتظار، برای ورودی ۴ به این برنامه خط «**blah! X is odd**» چاپ می‌شود. دلیل امر را نیز می‌تواند با کپی کردن این کد در ادیتوری که قابلیت **Auto Indent** دارد کشف کنید.

و نهایتاً مثال سوم می‌توان کد ساده‌ی زیر باشد

```

#include <iostream>
using namespace std;
int main()
{
    int x = 100;
    int y = 101;
    cout << x + y << endl;
    return 0;
}

```

که برخلاف تصوّر مقدار ۱۱۰ را چاپ می‌کند! دلیل امر را ادیتورهای «خیلی باهوش» که ارقام (۰ تا ۹) را به‌رنجی متمایز با حروف انگلیسی (خصوصاً حرف **l** کوچک) در می‌آورند، نشان می‌دهند.

با این ۳ مثال اهمیت انتخاب ادیتور مناسب نیز روشن شد. علاوه بر امکانات بالقوه ادیتور، تسلّط برنامه‌نویس بر ادیتور و توانایی کار متمرکز با آن (و آگاهی از نحوه انجام امکاناتی نظیر **copy-pase**، **indent-block** و **comment-multiple-line**) نیز شرط مهمی در انتخاب ادیتور است.

۴.۳.۱.۴ تکنیک سوم، انتخاب دیباگر مناسب

آشنایی با یک دیباگر مناسب (نظیر **gdb** یا دیباگر داخلی **Visual C++**) حتی در سطح مقدماتی می‌تواند در بسیاری از مواقع (مخصوصاً در کدهای بازگشتی) سودمند باشند. چنین دیباگرهای مجزایی امکان تعیین محل رخ دادن خطا (خصوصاً در مواقع **crash** یا **Segmentation Fault** برنامه) را بسیار ساده‌تر از روش نوشتن توضیحات اضافه فراهم می‌آورند.

۴.۳.۱.۵ تکنیک چهارم، انتخاب زمان و مکان مناسب!

می‌دانیم اکثر خطاهای برنامه‌نویسی ناشی از طراحی نادرست است؛ اما در پروژه‌های کوچک که عملاً فاز طراحی نیاز نیست (وجود ندارد)، عمده خطاهای برنامه‌نویسی از کجا ناشی می‌شوند؟

برای پاسخ دادن به این سؤال می‌توان اولین برخورد برنامه‌نویس پس از کشف یک خطای غیرعمدی‌اش را و زمان و شرایطی که آن خطا را ناخواسته انجام داده بررسی کرد. دور از ذهن نیست اگر ادعا کنیم که بیش‌تر خطاهای ناخواسته‌ی برنامه‌نویسی (خصوصاً در پروژه‌های فاقد طراحی) در شرایطی اتفاق می‌افتد که تمرکز و حضور ذهن برنامه‌نویس (که طراحی و کدنویسی را هم‌زمان انجام می‌دهد) درگیر مخاطره می‌شود.

انتخاب مکان مناسب (ترجیحاً ساکت یا بسته به عادت برنامه‌نویس، با صدا یا آهنگی خاص) و زمان مناسب (بسته به شناخت برنامه‌نویس از خودش، می‌تواند هر ساعتی از روز یا شب باشد)، در تمرکز و حضور ذهن برنامه‌نویس بسیار حائز اهمیت است. برخی از برنامه‌نویسان شب را (به دلیل آرامش و سکوت) ترجیح می‌دهد و برخی دیگر معتقدند که بازه فکری‌شان در ساعات اولیه‌ی صبح و پس از صرف صبحانه سبک حداکثر است.

در هر حال باید توجه داشت که شرایط محیطی، خصوصاً در پروژه‌های کوچک که استقلال و آزادی برنامه‌نویس بیش‌تر است می‌تواند از فاکتورهای اجتناب از خطا به‌شمار بروند. حتی در بسیاری از مواقع، ترک کردن یک کد خطا‌دار (که خطای آن علی‌رغم تلاش‌های زیاد کشف نشده) برای چند ساعت و یک استراحت موقت می‌تواند باعث شود تا ذهنیت‌های نادرست برنامه‌نویس (مثلاً بدیهی شمردن عملکرد یک دستور یا یک تابع) دچار تردید شده و راه برای کشف خطا باز شود. این نکته بارها برای تهیه‌کنندگان این مقاله اتفاق افتاده است.

نهایتاً دقت کنید که ۳ ساعت کار پربازده، بیش‌تر از ۱۰ ساعت کار کم‌بازده (که احتمالاً به چندین ساعت وقت برای **debug** احتیاج دارد) ارزش دارد. پس شاید بهتر باشد اگر ۱۰ ساعت برای انجام یک پروژه وقت دارید، ۳ ساعت آن را استراحت کنید، ۱ ساعت آن را حمام کرده و تلویزیون تماشا کنید، ۱ ساعت غذا یا دسر بخورید، ۲ ساعت در اینترنت بگردید و تفریح کنید و نهایتاً در ۳ ساعت با بازده عالی پروژه را تمام کنید! (البته از مدیریت ریسک غافل نشوید - مثلاً ممکن‌ست در ساعات آخر برق برود!).

۴.۳.۱.۶ تکنیک پنجم، درک درست صورت مسئله

نهایتاً آخرین اشتباه «مستقل از کد» رایج بین برنامه‌نویسان جوان را می‌توان «درک نادرست از صورت مسئله» بر شمرد. از آن‌جا که پروژه‌های کوچک فاز طراحی مستقل ندارند و حتی اگر هم داشته باشند، غالباً طراحی آن‌ها تست ندارد، تمام خطاهایی در شرایط عادی در فاز تحلیل و طراحی اتفاق می‌افتند، این‌بار به‌سوی کد سرازیر می‌شوند.

از آن‌جا که تشخیص و تصحیح خطا در کد نوشته شده هزینه‌ی زیادی می‌برد (مخصوصاً اگر معلوم شود که خطا مثلاً در الگوریتم بوده و کد باید به‌کل عوض شود)، پیش‌نهاد می‌شود حتماً پیش از شروع به کد زدن، مثال‌هایی از اجرای نمونه (خصوصاً در مسائل آموزشی/الگوریتمی که امکان تست میسرتر است) را روی الگوریتم پیاده کرده و از صحت اجرای آن مطمئن شوید.

۴.۳.۲ گروه اول، تکنیک‌های عمومی

۴.۳.۲.۱ تکنیک اول، جداسازی خروجی‌ها

چنان‌چه در تعریف «تشخیص» و تمایز آن با «پیش‌گیری» گفته شد، در بسیاری از موارد برنامه‌نویس تمایل دارد تا در صورت اتفاق افتادن شرایط خاص (خطاخیز) بدون لطمه به جریان عادی برنامه، او از این مسئله آگاه شود. بالطبع آگاهی از نحوه جریان برنامه همواره می‌تواند جزء اطلاعات مفید برای برنامه‌نویس، خصوصاً پیش از تحویل و در هنگام تهیه و تست، باشد.

متأسفانه اکثر برنامه‌نویسان جوان تنها با یک جریان (Stream) خروجی به‌نام `stdout` یا خروجی استاندارد آگاهی دارند و در هر شرایطی هر نوع خروجی (اعم از توضیحات و `debug info`) را به همان خروجی می‌فرستند. حال آن‌که در یک برنامه به‌صورت طبیعی دو جریان برای خروجی وجود دارد که هم از جانب سیستم‌عامل (ویندوز، لینوکس و ...) و هم از جانب زبان‌های برنامه‌نویسی (C، C++، جاوا و ...) پشتیبانی می‌شود. این جریان که `stderr` یا خروجی اشتباهات نام دارد در زبان C++ به‌صورت زیر استفاده می‌شود:

```
int main() {  
    cerr << "Program Started" << endl;  
    MyFunction(); // a sample function  
    cerr << "Program Terminated" << endl;  
    return 0;  
}
```

با افزودن همین ۲ خط ساده، اولاً از شروع و پایان برنامه اطمینان پیاده خواهیم کرد؛ هم عملاً در خروجی استاندارد چیزی نخواهیم نوشت.

لازم به یادآوری/آگاهی است که در محیط‌های ویندوز و لینوکس به‌طور مشابه می‌توان با استفاده از کاراکتر `>` و `>2` در انتهای نام فایل اجرایی، به‌ترتیب خروجی و خروجی اشتباهات را در یک فایل ریخت. برای مثال دستور زیر در محیط `command line` برنامه‌ی اجرایی `a.exe` را اجرا کرده و خروجی عادی (`stdout`) آن را در `out.txt` و خروجی اشتباهات آن را در `err.log` می‌ریزد:

```
C:\> a.exe > out.txt 2> err.log
```

برخی مزایای استفاده از دو جریان **err** و **out** عبارتند از:

- **کنترل کردن مجزای خروجی‌های برنامه و توضیحات ما:** برای مثال می‌توانیم در شرایط لازم (مثلاً هنگامی که خروجی عادی برنامه زیاد است) تنها توضیحات خودمان (**cerr** ها) را ببینیم و خروجی را در یک فایل بریزیم (با کمک **a.exe > out.txt**) و یا در هنگام تست، آن‌ها را در خروجی عادی نبینیم (**a.exe > err.log**).
- **خروجی درست برنامه، حتی در صورت فراموش کردن پاک کردن توضیحات:** بارها اتفاق می‌افتد که برنامه‌نویس به‌هنگام اشکال‌زدایی (**debug**) تعداد زیادی توضیحات (نظیر مقادیر متغیر در خط‌های خاص) می‌نویسد و در هنگام تحویل (**Release**) فراموش می‌کند یکی از این توضیحات را پاک کند و در عمل، استفاده‌کننده با خروجی نادرست و اضافی مواجه می‌شود حال آن‌که وجود توضیحات ناخواسته در خروجی اشتباهات، تأثیری در خروجی برنامه ندارد. در تمرینات و مسابقات برنامه‌نویسی (نظیر **ACM ICPC**)، این غفلت ممکن‌ست موجب ساعت‌ها اتلاف وقت برای پیدا کردن خطای برنامه‌ای بشود که یک توضیح اضافه در خروجی عادی می‌نویسد!
- **امکان تحلیل و مشاهده‌ی رفتار برنامه حتی بعد از شکست:** برنامه‌ای را در نظر بگیرید که به‌ازای یک ورودی خاص، در حلقه‌ی نامتناهی نوشتن یک خط خاص به‌دام می‌افتد و الزاماً باید توسط کاربر متوقف شود (مثلاً با **Ctrl+C**). نوشتن توضیحات در خروجی عادی (لابه‌لای خطوط بی‌نهایت) معمولاً کمک دشواری به پی‌گیری اجرا می‌کند، حال آن‌که با نوشتن توضیحات اضافه در خروجی اشتباهات، می‌توان مستقل از خروجی عادی به رفتار کد پی برد. این مورد در برنامه‌های **Interactive** و برنامه‌هایی که منجر به از کار افتادگی (**hang up**) سیستم می‌شود با ریختن خروجی اشتباهات در یک فایل (**a.exe 2> err.log**) بسیار مفید خواهد بود.
- **افزایش خوانایی برنامه:** خواننده‌ی کد (که می‌تواند صرفاً همان نویسنده‌ی کد باشد) با مشاهده‌ی رشته (مقدار) ای که در خروجی اشتباهات نوشته می‌شود، می‌تواند پی ببرد که این مهم جزو خروجی نیست و به‌راحتی از آن می‌تواند بگذرد؛ اما در صورتی که همه چیز در خروجی استانده نوشته شود، فهم این نکته تنها از طریق مفهومی و آگاهی از خروجی خواسته شده امکان پذیر است. برای مثال ۲ کد زیر را (از نظر مفهومی و به‌عنوان اولین دید) مقایسه کنید:

```
for (int i=0; i<n; i++) {  
    sum += a[i];  
    cout << a[i] << endl;  
}  
cout << sum << endl;
```

```
for (int i=0; i<n; i++) {  
    sum += a[i];  
    cerr << a[i] << endl;  
}  
cout << sum << endl;
```

در انتهای توضیحات این تکنیک می‌توان اشاره کرد که در برخی مواقع تعریف چند جریان خروجی اضافه (**ostream**) که برای مثال در چند فایل جدا اتفاقات را بنویسند و یا حتی تعدادی جریان ورودی (که عملاً توقف در برخی شرایط را موجب شده و امکان کنترل صریح‌تر را به برنامه بدهند) می‌تواند بسیار مفید باشد.

۴.۳.۲.۲ تکنیک دوم، چاپ سریع متغیرها

ماکروی زیر را در نظر بگیرید:

```
#define show(x) cerr << #x << " = " << x << endl;
```

با کمک این ماکرو، به سادگی و در هر کجای برنامه می توان با نوشتن دستور ساده‌ی `show(var)` مقدار متغیر `var` را در خروجی اشتباهات مشاهده کرد. ابزار `#` پیش از نام یک متغیر، نام رشته‌ای آن را جایگزین می کند تا خروجی کد زیر عیناً `var = 20` باشد.

```
int var = 400 / 20;  
show(var);
```

۴.۳.۲.۳ تکنیک سوم، کامنت (و آن کامنت) کردن سریع یک تکه از کد

کد زیر را در نظر بگیرید:

```
cout << "I'm always in." << endl;  
/**/  
cout << "I needed to be toggled!" << endl;  
MyFunc();  
/**/  
Cout << "I'm always in." << endl;
```

در این کد تنها با برداشتن و گذاشتن / قرمز رنگ، دو خط پایینی آن کامنت شده و یا از حالت کامنت بودن خارج می شوند. با کمک این تکنیک می توانید یک تکه از کد را که احتیاج به کامنت و آن کامنت کردن متوالی دارد، مدیریت کنید.

۴.۳.۲.۴ تکنیک چهارم، پایان پروژه چند ساعت پس از پایان پروژه است!

هرگز به بهانه‌ی نهایی شدن پروژه توضیحات اضافی خود را پاک نکنید! دقت کنید که جداسازی کامنت‌ها از متن برنامه در زمان کامپایل انجام می شود و حجم کامنت‌ها هیچ تأثیری در زمان و نحوه‌ی اجرای کد ندارد. از سوی دیگر علی‌رغم این که برنامه‌نویسان به زیبایی و تمیزی کد اهمیت می دهند، خوبست به یاد داشته باشند که توضیحات اضافی (نظیر `debug info`) که کامنت شده باشد، به صورت رایگان مستندسازی را برای پروژه بهارمغان آورده‌اند!

پس هرگز چیزی را از کدتان پاک نکنید مگر حداقل چند ساعتی از پایان پروژه و تحویل آن (و حتی گرفتن نمره/حق الزحمه) گذشته باشد! توجه داشته باشید که یک کد غلط (که بالفرض پیاده‌سازی یک الگوریتم غلط است) می تواند با تغییر اندکی به یک کد درست (پیاده‌سازی الگوریتم درست) تبدیل شود.

در صورتی که نیازی به یک بخش از کد ندارید و حس می کنید فضای مورد استفاده در طول صدها خط کد را اشغال کرده است، به جای پاک کردن آن تکه از کد، آن را در فایل دیگری (و حتی در پوشه‌ی دیگری) نگهداری کنید.

۴.۳.۳ گروه دوم، پیش‌گیری از خطاهای اعمال شده خارجی

۴.۳.۳.۱ تکنیک اول، بررسی کردن تعاملات با سیستم

ممکن است که برنامه‌نویس فرض کند که هر دستور `int *p = new int;` با موفقیت اجرا می‌شود؛ مخصوصاً اگر حافظه‌ی RAM کامپیوتر وی 4 گیگابایت باشد! اما باید توجه داشت که طبق یک اصل ساده

هر دستوری که امکان *fail* شدن داشته باشد، ممکن است *fail* شود؛ حتی اگر احتمال آن بسیار کم باشد.

با این وصف، بررسی `null` نبودن هر متغیری که به‌صورت پویا (*dynamic*) تخصیص می‌یابد یک راه ساده و کم‌هزینه‌ی پیش‌گیری از خطاست. مشابه همین کار در تعاملات I/O (نظیر بازکردن یک فایل متنی) است که می‌تواند به‌راحتی پس از انجام دستور بررسی شده و از شکست سیستم جلوگیری کند. دقت کنید که هزینه‌ی (زمانی) چک کردن `null` بودن یک متغیر (یا منبع I/O) تنها یک بار (بلافاصله بعد از تخصیص) وارد عمل می‌شود و در همین یک‌بار نیز، این هزینه بسیار کم‌ترین از هزینه‌ی تخصیص (و حتی شکست) است. پس تنها دلیل توجیه‌پذیر برای انجام ندادن این چک، تنبلی برنامه‌نویس است!

۴.۳.۳.۲ تکنیک دوم، تست‌ها، بررسی و تست‌های بی‌جواب

در صورتی که به برنامه‌ی خود تستی می‌دهید و رفتار نامناسب از آن مشاهده می‌کنید، اولین برخورد شما می‌تواند بررسی خود تست‌ها باشد. دقت کنید که تست‌ها نیز کدهای ساده‌ای (نه لزوماً به زبان C++) هستند که امکان وجود خطا در طراحی، پیاده‌سازی و حتی تایپ آن‌ها وجود دارد!

البته یکی از تکنیک‌های بسیار مطلوب پیش‌گیری از خطا، دادن تست‌های ورودی بزرگ بدون دانستن جواب درست آن‌هاست. دقت کنید که شما ممکن است ندانید میانگین ۱۰۰۰ عدد تصادفی بین صفر تا ۱۰۰۰ چند می‌شود اما می‌دانید که یک برنامه‌ی درست که برای این منظور نوشته شده باشد، اولاً در مدت نسبتاً کمی باید پایان یافته و یک عدد چاپ کند. ثانیاً این عدد باید خودش بین صفر تا ۱۰۰۰ باشد. ثالثاً این عدد نباید بیش از ۳ رقم اعشار رقم معنی‌دار داشته باشد.

۴.۳.۳.۳ تکنیک سوم، هیچ‌چیز ۱۰۰٪ درست نیست!

هرگز از تست کردن ورودی غافل نشوید. برای مثال در یک تمرین دانشگاهی، علی‌رغم ادعا و تأکید ورودی دهنده، ممکن است ورودی برنامه غلط باشد و تعدادی از برنامه‌های مشابه برنامه‌ی شما، تصادفاً با این ورودی‌های غلط درست کار کنند و باعث شوند هرگز کسی نفهمد که سیستم اشتباه کار می‌کند. سعی کنید تا حد امکان هر نوع داده‌ی خارجی که به سیستم شما وارد می‌شود را از یک فیلتر تست عبور دهید تا بعداً به‌راحتی از خطادار نبودن آن‌ها اطمینان داشته باشید و در صورت شکست سیستم، بتوانید با تمرکز بیش‌تر روی کد خودتان متمرکز شوید.

سعی کنید با یافتن خطای ورودی، حتی المقدور رفتاری نشان دهید که آگاهی تست‌کننده را به‌همراه داشته باشد. برای مثال با نوشتن «Invalid input» و پایان برنامه ممکن است تست‌کننده این رشته را به‌عنوان جواب شما تلقی کرده و برنامه‌ی شما را «خروجی نادرست» قلمداد کند، حال آن‌که انداختن برنامه در یک حلقه‌ی نامتناهی، دسترسی به منابع غیر مجاز (نظیر reboot کردن سیستم، در صورتی که نتایج بدی برای شما نداشته باشد!) یا crash کردن درونی (مثلاً تقسیم بر صفر) احتمال بیش‌تری برای آگاهی تست‌کننده دارند.

۴.۳.۴ گروه سوم، پیش‌گیری و تشخیص خطاهای درون‌کد

۴.۳.۴.۱ تکنیک اول، استفاده از ثابت‌ها (const)

استفاده از توابع، نه‌تنها امکان پارامتریزه کردن کد (برای حداکثر مقادیر ورودی یا یک عدد مشخص شده و قابل تغییر در راستای scalability) را فراهم می‌آورند، بلکه می‌توانند در پیش‌گیری یکی از خطاهای رایج سهم زیادی داشته باشند.

فرض کنید در برنامه‌ای، حداکثر ساینز ورودی 100000 نوشته شده و برنامه‌نویس اشتباهاً آن را 10000 می‌خواند و در ابتدا، وسط و داخل یکی از توابع، ۳ آرایه با این اندازه تعریف می‌کند و در داخل آن تابع نیز یک حلقه‌ی for از 0 تا 9999 روی عناصر آرایه زده است. پیش از تحویل پروژه، برنامه‌نویس به‌صورت تصادفی می‌فهمد که عدد درست 100000 بوده و او یک صفر را کم خوانده است. او بلافاصله به کدش بر می‌گردد و با یک دستور find در ادیتورش، ۳ محل تعریف آرایه‌ها را تصحیح می‌کند اما متوجه حلقه‌ی for نمی‌شود...

۴.۳.۴.۲ تکنیک دوم، همه‌ی ثابت‌ها int نیستند

در زبان C++ شما می‌توانید مقادیری نظیر 0XFF یا 0777 را (که به‌ترتیب در مبنای شانزده و هشت هستند) به متغیرها انتساب کنید. جدای از این مورد، برای نوشتن برخی اعداد ناخوانا (نظیر 1000000) می‌توانید آن‌ها را به‌صورت ضرب دو عدد خوانا (مثلاً 1000*1000) نوشته و با این کار از مقدار درست آن‌ها اطلاع پیدا کنید.

هم‌چنین در مورد اندازه‌ی آرایه‌ها، شما می‌توانید آن‌ها را به‌صورت جمع یا ضرب یا فرمولی از سایر ثابت‌ها بنویسید.

۴.۳.۴.۳ تکنیک سوم، تعریف مجدد تایپ‌های ساده (نظیر *int*) به نام‌های مفهوم‌دار

کد زیر را از نظر خوانایی در نظر بگیرید

```
typedef string NameType;
typedef int SalaryType;
typedef int EmployeeID;

map<EmployeeID, SalaryType> m;
```

چنانچه مشاهده می‌شود، با تعریف تایپ‌های اولیه به نام‌های مفهوم‌دار، هم کد را خودمستند (Self-Documented) کرده‌ایم، هم امکان تغییرات آتی را فراهم آورده‌ایم (مثلاً تبدیل تایپ *EmployeeID* به *long long* یا *string*) و هم از اشتباهاتی نظیر جابه‌جا گرفتن عنصر *first* و *second* در *map* (که هر دو *int* هستند) پیش‌گیری کرده‌ایم.

۴.۳.۴.۴ تکنیک چهارم، تعریف متغیرها، هر چه نزدیک‌تر به محل استفاده

هرچه قدر *scope* آزاد بودن یک متغیر بالا باشد، احتمال خطای آن بیش‌تر است. مثال زیر را ببینید:

```
void func() {
    int guilli, guilli;
    guilli = 2;
    // something on guilli again
    Print(guilli);

    guilli = 3;
    // something on guilli, now
}
```

چنان‌که می‌بینید، تعریف زودهنگام *guilli* (نسبت به محل مصرف آن برای اولین بار) باعث شده تا خطای تایپی موجود در *Print* تشخیص داده نشود.

با استدلال و مثالی مشابه، می‌توان گفت که متغیرهای *Global* نیز خطاخیزی برنامه را افزایش داده و از اتکاپذیری آن می‌کاهند. متغیرهای *static* یک *struct* (یا *class*) و بعضاً ارسال آرایه‌ها به توابع می‌توانند جایگزین‌هایی برای متغیرهای *Global* باشند.

۴.۳.۴.۵ تکنیک پنجم، پرهیز از ارسال با مرجع (*reference*) به پارامترهای تابع، مگر با کمک *const*

متأسفانه اکثر برنامه‌نویسان به‌طور ضمنی عادت دارند پارامترهای ارسالی به توابع را تغییر داده و عملیات را روی خود آن‌ها انجام دهند. این امر در زمانی که (در راستای افزایش سرعت و *performance*) برخی اشیاء سنگین به صورت *by-reference* به توابع فرستاده می‌شوند، می‌تواند خطرات جدی به‌وجود بیاورد.

در چنین مواقعی، `const &` کردن این پارامترها، شیء ارسالی را از تغییر بیمه می‌کند.

۴.۳.۴.۶ تکنیک ششم، حداکثر استفاده از توابع و ساختارهای خود زبان

زبان `C++` دارای کتابخانه‌های قدرتمند قدیمی به ارث رسیده از `C` (نظیر `cmath`، `cctype`، `cstdlib` و ...) و نیز کتابخانه‌ی پر قدرت `STL` (Standard Template Library) است. در این کتابخانه‌ها تعداد زیادی از توابع رایج و پر استفاده به نحو مطلوب (سریع) و مطمئنی آماده‌ی استفاده قرار گرفته‌اند که جدا از صرفه‌جویی برنامه‌نویس در زمان (عدم نیاز به نوشتن مجدد)، اطمینان وی از صحت عمل کرد و سرعت کار را نیز تضمین می‌کنند.

برای مثال توابع `sort` یا `next_permutation` در `STL` را در نظر بگیرید که در صورتی که بخواهد توسط یک برنامه‌نویس مبتدی نوشته‌شدند یا سرعت خود را از دست می‌دهند یا احتمال خطا در آن‌ها بالا می‌رود.

به‌عنوان مثال در یکی از آزمون‌های برنامه‌سازی سال گذشته که در دانشکده برگزار شد، عنوان شده بود که در ابتدا و انتهای رشته‌ی ورودی (در یک خط) می‌تواند فاصله‌ی خالی وجود داشته باشد و یکی از مؤلفان این مقاله این مهم را با `if (s[i] == ' ')` بررسی کرده بود که کاراکترهای `'\t'` را از نظر نمی‌گذارند! حال آن‌که با کمک تابع `isspace(s[i])` از کتابخانه‌ی `cctype`، این بررسی مطمئن‌تر و بدون نقص انجام می‌شد.

۴.۳.۴.۷ تکنیک هفتم، هر حلقه‌ای پایان می‌یابد

کد زیر را در نظر بگیرید که به جای حلقه‌ی `while (true)` از حلقه‌ی `for` استفاده کرده و به لحاظ معنایی می‌داند که این حلقه بیش از ۱۰۰۰۰ بار نباید اجرا بشود.

```
int i = 0;
for (; i <= 10*1000; i++) {
    // normal in-while-true stuffs, here
}
if (i == 10*1000) {
    Print("Infinite Loop prevented");
}
```

علاوه بر جلوگیری از حلقه‌ی نامتناهی در این کد (که ممکن است حتی به `reboot` شدن سیستم بیانجامد)، رخ دادن چنین حلقه‌ای تشخیص داده شده و به اطلاع کاربر می‌رسد.

مشابه همین تکنیک را می‌توان با ارسال تنها یک پارامتر اضافه (نظیر عمق) به توابع بازگشتی برای جلوگیری از بازگشت نامتناهی (`infinite recursion`) این توابع انجام داد.

۴.۳.۴.۸ تکنیک هشتم، پرهیز از گزینه‌های منوی *Edit* نظیر *copy* و *paste* و نیز *replace*

Copy و **Paste** را اصطلاحاً مادر همه‌ی **bug**های برنامه‌نویسی می‌نامند! علت این امر نیز نادیده گرفتن تغییرات در هنگام **Paste** و یا بعداً (هنگامی که یکی از دو قطعه‌ی شبیه هم ویرایش می‌شود و دیگری نمی‌شود) است. از این رو مناسب‌ست که حدالمقدور از **Copy** و **Paste** استفاده نکرده و با تعریف تابع، یک **soft-copy-paste** ایجاد کنیم.

با گرفتن تابع در مواقع مورد نیاز، علاوه بر یک‌سان سازی تغییرات آتی، طول کد و توابع دیگر کاهش یافته و تسلط برنامه‌نویس بر پروژه بیش‌تر می‌شود.

هم‌چنین به‌هنگام **replace** کردن نام یک متغیر، ممکن‌ست متغیرهایی که نام مشابه (و حتی پیش‌وندی/پس‌وندی) دارند دست‌خوش تغییر شوند. برای اجتناب از این مسئله، توصیه می‌شود نام‌های متغیرها را حتی‌الامکان منفصل گرفته و **replace** را (در صورت لزوم) با بررسی تک به تک انجام دهیم.

۴.۳.۴.۹ تکنیک نهم، پرهیز از خروج با *exit()*

مطلوب‌ست یک برنامه همواره از نقطه‌ی پایان **main** خارج شود و تا حد امکان از استفاده از **exit()** (مگر در مواقعی جلوگیری از شکست) اجتناب شود. دو علت اصلی برای این امر عبارتند از:

۱- سناریویی را در نظر بگیرید که شما برنامه‌ای برای یک ورودی نوشته‌اید و نگاه ورودی مسئله به «چند سری تست در یک پاس ورودی» تغییر می‌یابد. در این حالت شما از بعد از شروع **main()** تا پیش از **return 0** انتهای آن در یک حلقه (ی مناسب) قرار می‌دهد اما فراموش می‌کنید که ممکن‌ست در اثنای اجرای حلقه، یکی از **exit** های شما (که پایان یک تست مفرد را در بر داشته‌اند) فراخوانده شده و عملاً کارکرد سیستم را مختل کند.

۲- سناریویی را در نظر بگیرید که شما برنامه‌ای نوشته‌اید، منتها پس از پایان کد (و استفاده از **exit()** در خلال کد) متوجه می‌شوید که در صورت مسئله از شما خواسته شده که در انتهای کد یک اطلاع دیگر (مثلاً نام برنامه‌نویس) را نیز چاپ کند. در این حالت شما به آخرین خط **main** رفته و پیش از **return 0** آن، خط چاپ اطلاع جدید را چاپ می‌کنید، حال آن‌که در خروجی‌های ناشی از **exit()** این خط نوشته نمی‌شود.

۴.۳.۴.۱۰ تکنیک دهم، نام‌گذاری سازگار (*consistent*) و با معنی

انتخاب نام‌های بامعنی و مرتبط و نیز سازگار با خودتان (مثلاً همیشه **g** برای ماتریس مجاورت گراف، یا **r** برای یک **reference**) اکیداً احتمال پیدایش خطاهای ناشی از برنامه‌سازی و نام‌گذاری را کاهش می‌دهد.

توصیه می‌شود که نام‌ها را طوری انتخاب کنید که (به‌جز متغیرهای واضح) حداقل ۳ حرف با یک نگارش خوانا و ترجیحاً به زبان انگلیسی داشته باشید تا **self-documented** بودن کد شما را افزایش دهد.

۴.۳.۴.۱۱ تکنیک یازدهم، مقداردهی مقادیر بی‌ربط به‌عنوان مقادیر اولیه

برنامه‌ی زیر را در نظر بگیرید که قرار است ۳ عدد را از ورودی خوانده و جمع آن‌ها را حساب کند.

```
int a[4];
for (int i=1; i<=3; i++)
    cin >> a[i];
int b = 0;
for (int i=0; i<=3; i++)
    b += a[i];
cout << a[i];
```

در دومین حلقه‌ی این کد، برنامه‌نویس اشتباهاً حلقه‌ی را از ۰ شروع کند و مقدار $a[0]$ نیز در مجموع محاسبه می‌شود.

واضح‌ست که مادامی که مقادیر اولیه‌ی آرایه‌ی a صفر است برنامه‌نویس از این خطر آگاه نخواهد شد، اما اگر مقادیر اولیه‌ی عناصر این آرایه با یک عدد عجیب (مثلاً ۱۰۰۰,۰۰۰-) مقداردهی شوند، در اولین تست‌های برنامه، به احتمال زیاد برنامه‌نویس از این موضوع آگاه خواهد شد.

۴.۳.۴.۱۲ تکنیک دوازدهم، قانون ساده‌ی اولویت عملگرها

کد زیر را در نظر بگیرید:

```
int x, y;    // Two numbers
x = 1;
y = x<<2 + 1; // x<<2 = 4 so y = 4+1 = 5
cout << "Y=" << y << endl;
```

خروجی این کد برخلاف تصوّر ۵ بوده و دلیل آن نیز اولویت $+$ بر $<<$ می‌باشد. یک قانون ساده‌ی اولویت عملگرها می‌گوید

۱- اولویت $*$ و $/$ از $+$ و $-$ بیش‌ترست.

۲- عملگرهای هم‌اولویت از چپ به راست محاسبه می‌شوند.

۳- اولویت پرانتز از همه‌چیز بیش‌تر است.

۴- سایر اولویت‌ها را فراموش کن!

۴.۳.۴.۱۳ تکنیک سیزدهم، صرفه‌جویی نکردن بی‌مورد

کد زیر را در نظر بگیرید که قوارست ۳ عدد را خوانده و مجموع آن‌ها را حساب کند.

```
#include <iostream>
using namespace std;

int a[3], b;

int main() {
    b = 0;
    for (int i=1; i<=3; i++) {
        cin >> a[i];
        b = b + a[i];
    }
    cout << b << endl;
}
```

این کد به ازای ورودی 4 5 6 مقدار 12 را چاپ می‌کند. (چرا؟)

در هنگام تعریف آرایه‌ها، هرگز صرفه‌جویی (خساست!) به‌خرج ندهید و همواره سائز آرایه‌ها را حداقل ۲ واحد بزرگ‌تر از مقدار

مورد نیاز بگیرید.

۴.۳.۵ گروه چهارم و آخر، تکنیک‌های نهایی

۴.۳.۵.۱ تکنیک اول، برداشت صحیح از زمان اجرا

قرار دادن برخی تکنیک‌های گفته‌شده در راستای بهبود پیش‌گیری از خطا ممکن‌ست باعث افزایش زمان اجرا بشوند اما درک درست زمان اجرای برنامه می‌تواند اطمینان شما از ناچیز بودن این افزایش را سبب شوند. برای مثال زمان لازم برای بررسی `null` نبودن پوینتر به یک فایل پس از باز کردن آن در مقایسه با زمان مورد نیاز برای عملیات I/O شناسایی و باز کردن فایل عملاً ناچیز شمرده می‌شود.

دقت کنید که کدی که ۲/۰۰۰۰۱ ثانیه طول می‌کشد و اتکاپذیری آن ۹۸٪ است بسیار بهتر از کدی است که ۲ ثانیه طول می‌کشد و اتکاپذیر آن ۸۰٪ است!

۴.۳.۵.۲ تکنیک دوم، کثیف رسیدن بهتر از هرگز نرسیدن است!

علی‌رغم این‌که یک کد زیبا ممکن‌ست برای شما (به‌عنوان برنامه‌نویس) بسیار جذاب و حائز اهمیت باشد اما توجه داشته باشید که برنامه‌ی شما در صورتی که وظیفه‌ی خواسته شده را به‌درستی و با اطمینان انجام ندهد، زیباتر و جذاب‌تر از آثار شکسپیر، نقاشی‌های پیکاسو یا اشعار مولانا نخواهد بود!

۴.۳.۵.۳ تکنیک سوم، تنبل نباشید!

دیر رسیدن بهتر از هرگز نرسیدن است!

یک پروژه‌ی نادرست (که بعد از رفع یک اشکال کوچک سریع‌ترین و تمیزترین و کوچک‌ترین پروژه‌ی صورت مسئله باشد)، به‌واسطه‌ی همان یک اشکال کوچک، بی‌ارزش‌ترین از یک پروژه‌ی کند و کثیف و طولانی است! پس بین «درست کار کردن پروژه»، «سرعت اجرای پروژه»، «سرعت تایپ پروژه»، «تعداد خط‌های پروژه» و نهایتاً «تمیزی، ظرافت، خلاقیت و زیبایی پروژه» اولویت بندی درستی برقرار کنید.

۵ جمع‌بندی

Casey Charlton در یکی از مقالات خود [6] و نیز Kirk Knoernschild (نویسنده‌ی کتاب‌های مرجع انتشارات Addison-Wesley) در مقاله‌ای دیگر [7] اشاره می‌کند که ویژگی‌های اصلی یک کد خوب (و بالطبع اتکاپذیر) در ذات به شرح زیرند. واضح‌ست که رعایت این ویژگی‌ها ساده‌ترین و طبیعی‌ترین روش ارتقاء اتکاپذیری (از نوع پیش‌گیری) هستند.

• سادگی:

بدون شک سادگی شرط اول یک کد خوب است. حتی مسائل پیچیده‌ی علوم مختلف نیز با کدهای ساده قابل حل هستند و ابزارهای جدید برنامه‌سازی (و آزادی‌هایی که آن‌ها به ارمغان می‌آورند)، صرفاً برای سهولت کار برنامه‌نویس ابداع شده‌اند، نه برای پیچیده‌تر کردن ماجرا. یک برنامه‌نویس خوب باید همیشه جمله‌ی معروف آلبرت انیشتین را به‌خاطر داشته باشد:

Make everything as simple as possible, but not simpler

• کارکرد (Performance) مناسب، مطابق آنچه انتظار می‌رود:

برای مثال اعمال روش‌های گفته‌شده در این مقاله نباید از کارکرد کد بکاهد. برعکس، روش‌های گفته‌شده در راستای ارتقاء اتکاپذیری می‌بایست به بهبود کارکرد کد خوب کمک کنند.

• تغییر پذیری:

حتی اگر تنها خواننده‌ی کد، خود نویسنده‌ی آن باشد، باید همیشه در نظر داشته باشد که حافظه و آگاهی و دانش او در چند سال بعد، از او آدم دیگری می‌سازد که تکنیک‌های عجیب و نامتعارف کد حال حاضر وی را نخواهد شناخت. یک کد خوب همواره باید طوری نوشته شود که قابلیت هرگونه تغییر پذیری را در هر زمانی و توسط هر برنامه‌نویس توانایی ممکن (و بالطبع، اندکی سخت) سازد.

• خوانایی

یک کد خوانا مانند یک شعر (یا قطعه ادبی) باید از بالا تا پایین به سادگی خوانده شود و هیچ برآمدگی‌ای نداشته باشد. بالطبع هزینه‌ی نوشتن چنین کدی برای بار اول بسیار بالاست که هزینه را خوانندگان آن کد، باید با لذتی که می‌برند، بپردازند!

و جمله‌ی آخر این که

برنامه‌نویسان ماهر، افرادی خلاق هستند نه لزوماً افرادی تکنیکال! تکنیک قابل یادگیری و انتقال است اما خلاقیت نیاز به شناخت و پرورش دارد و تمرین تنها راه دستیابی به این مهارت است.

۶ پیش‌نهاداتی برای کارهای آینده

چنان‌چه در فاز دوم (تعریف پروژه) اشاره شد، یکی از مخاطبان نهایی این مقاله، همانا دانش‌جویان سال‌های آغازین دانشگاه می‌باشند. با این وصف، لازم به ذکر است که به‌نظر نویسندگان این مقاله، تمامی نکات و تکنیک‌های گفته‌شده تنها زمانی مفید فایده خواهند بود که در عمل و به‌صورت ذاتی در برنامه‌نویس القاء شوند. بالطبع بررسی لحظه‌ای و بازبینی این نکات قبل (و در هنگام) کد زدن، جز دغدغه و نگرانی چیزی به‌ارمغان نخواهد آورد. از این رو، توصیه می‌شود که تمامی نکات گفته‌شده در این مقاله، صرفاً در عمل و به‌تدریج به برنامه‌نویسان جوان آموزش داده‌شود.

اما در صورت تمایل به ادامه‌ی کار در این زمینه، موضوعات و راهکارهای زیر پیش‌نهاد می‌شوند.

۱. به‌وضوح این مقاله تنها بخش بسیار کوچکی از تکنیک‌های ساده‌ی افزایش ارتقاءپذیری را شامل می‌شود و افزودن و جمع‌آوری تکنیک‌های بیش‌تر مفیدترین راهکار آتی به‌شمار می‌رود. در این راستا، نکته‌ی قابل توجه سادگی تکنیک و مقرون به‌صرفه بودن آن است - چه‌بسا تکنیک‌های مفیدی که هزینه (زمان، افزایش پیچیدگی) آن‌ها بیش‌تر از فایده‌شان است.
۲. نکات گفته‌شده در این مقاله، غالباً متناسب با زبان ++C بوده و علی‌رغم نزدیکی به سایر زبان‌های رایج (نظیر Java)، تکنیک‌های خاص آن زبان‌ها را شامل نمی‌شوند. جمع‌آوری این تکنیک‌های خاص علاوه بر تکمیل این مقاله می‌تواند به‌عنوان معیار جالبی برای سنجش «اتکاپذیری صرف» بین زبان‌های مختلف برنامه‌سازی استفاده شود.
۳. تحلیل و بررسی شکست‌های پروژه‌هایی که مشمول صفات «پروژه‌ی کوچک» می‌شوند می‌تواند آمار مناسبی برای تأکید در دسته‌های مختلف تکنیک‌ها ارائه دهد. برای مثال احتمالاً شکست‌های I/O شیوع کم‌تری و سبک‌تری نسبت به شکست‌های «توابع بازگشتی» دارند. با داشتن اطلاعاتی از این دست، می‌تواند سرمایه‌گذاری بیش‌تری در جست‌وجو برای تکنیک‌های مشکلات مختلف داشت.
۴. در این مقاله، بیش‌تر بر «پروژه‌های کوچک» از نوع آکادمیک و خاص منظوره تأکید شده است. کار بر روی پروژه‌های کوچک نوع سوم یعنی «گسترش‌پذیر» و بررسی انطباق تکنیک‌های گفته‌شده در این مقاله می‌تواند از راهکارهای سودمند برای ادامه باشد، چرا که نحوه‌ی نگرش و اهمیت «اتکاپذیری» در این دسته از پروژه‌ها متفاوت با دو دسته‌ی اول بوده و بعضاً تعدادی از ویژگی‌های (معلول) گفته‌شده را شامل نمی‌شوند.

- 1- Roger S. Pressman, "Software Engineering - A Practitioner's Approach", 5th Ed, 2001, McGraw Hill.
- 2- Musa, J.D., A. Iannino, and K. Okumoto, Engineering and Managing Software with Reliability Measures, McGraw-Hill, 1987.
- 3- Shigeo Shingo, Zero Quality Control: Source Inspection and the Poka-yoke System, Productivity Press, 1986.
- 4- Yuko Hirotsu, "A new method for preventing human errors", Central Research institute of Electric Power Industry Journal (CRIEPI News), September 2006, Article 432.
- 5- Adam Kolawa, "Why Aren't We Doing More to Prevent Errors?", Parasoft Corp. Available at www.computerworld.com/printthis/2003/0,4814,83735,00.html
- 6- <http://devlicio.us/blogs/casey/archive/2008/05/06/good-code-is.aspx>
- 7- <http://www.artima.com/weblogs/viewpost.jsp?thread=96404>
- 8- Steve Oualline, How Not to Program in C++: 111 Broken Programs and 3 Working Ones, or Why Does 2+2=5986, ACM 24x7 library, No Starch Press, 2003.