برنامه‌سازی پیشرفته

تمرین کامپیوتری امتیازی

مدرس: **رامتین خسروی**

طراحان: **پاشا براهیمی**، **شهریار عطار**، **علی عطاالهی**، **طاها فخاریان**

مهلت تحویل: یکشنبه ۱۳ فروردین ۱۴۰۲، ساعت ۲۳:۵۵

مقدمه

هدف این تمرین، تمیز کردن و بازآرایی[1] کدی است که به عنوان راه حل برای یک مسئله ارائه شده است؛ مسئله‌ی انتخابی، تمرین کامپیوتری سوم درس برنامه‌سازی پیشرفته در ترم پاییز ۱۴۰۰ است که صورت آن برای آشنایی بیشتر شما، در کنار صورت این تمرین آپلود می‌شود. در طول مراحل این تمرین، تلاش کنید تا ساختار کلی و طراحی کد اولیه و هم‌چنین درستی[2] آن را حفظ کنید. هم‌چنین، در این تمرین، با ابزار کنترل نسخه‌ی git نیز آشنا می‌شوید؛ در بخش معرفی git توضیح بیش‌تری برای این ابزار آمده است.

فرایند بازآرایی باید در مرحله‌های کوچک اجرا شود و پس از هر تغییر با اجرای موارد آزمون از درستی کد خود مطمئن شوید. هم‌چنین، پس از هر مرحله‌ی بازآرایی، بایستی تغییرات اعمال شده را به کمک ابزار گیت ذخیره کنید. علاوه بر توضیحاتی که در صورت پروژه در رابطه با ابزار گیت آمده است، توصیه می‌شود ویدیوی آشنایی با استفاده از گیت را که در بخش منابع بیشتر برای یادگیری گیت آورده شده است مشاهده کنید.

سه خلاصه‌ی مفید نیز در بخش خلاصه‌ها قرار داده شده است که می‌توانید از آن‌ها به عنوان راهنما استفاده کنید.

کد تمیز

در دنیای واقعی، در اکثر مواقع شما بعد از یک طراحی نسبتاً خوب و پیاده‌سازی آن، برای مدتی طولانی از آن کد برای هدف خود استفاده می‌کنید و در طول این مدت تغییراتی در آن ایجاد می‌کنید و قابلیت‌های زیادی را به آن می‌افزایید. پس از مدتی نه چندان طولانی، این تغییرات باعث می‌شوند که شما دیگر عملکرد کد را به وضوح متوجه نشوید و به تبع آن، توانایی شما برای تغییر و گسترش نرم‌افزار کاهش می‌یابد. این موضوع هنگامی تأثیر پررنگ‌تری پیدا می‌کند که برنامه‌نویسان متعددی طی مدت طولانی روی یک نرم‌افزار بزرگ کار می‌کنند که در شرکت‌های تولید نرم‌افزار امری عادی است. این زنجیره‌ی اتفاقات به ظاهر ساده وقتی به مرور زمان تکرار می‌شود، باعث بروز چالش‌های جدی در بسیاری از شرکت‌های نرم‌افزاری گشته و موجب بازماندن آنها از گردونه رقابت گشته است.

برای جلوگیری از این اتفاقات، توسعه‌دهندگان سعی می‌کنند که کدهای خود را تا حد امکان تمیز بنویسند. تعریف پرکاربردی که برای کد تمیز ارائه می‌شود، کدی است که به سادگی قابل فهم و تغییر دادن باشد؛ می‌توانیم آسان بودن فهم کد را معادل با آسان بودن فهم موارد زیر در نظر بگیریم:

- جریان اجرای برنامه

[1] refactoring

[2] correctness

- نقش و مسئولیت هر ساختار[3] (یا کلاس)
- هر تابع چه کاری انجام می‌دهد
- هر متغیر یا عبارت برای چه هدفی نوشته شده است

رعایت موارد بالا، گسترش و بازآرایی کد را ساده‌تر می‌کند؛ زیرا برای فردی که تغییرات را ایجاد می‌کند، کد قابل فهم است و می‌داند که این تغییرات، عملکرد[4] فعلی کد را تغییر نمی‌دهند.

شرح مسئله

در این تمرین، برای تمرین کامپیوتری سوم درس برنامه‌سازی پیشرفته در ترم پاییز ۱۴۰۰، یک راه حل ارائه شده است. این راه حل صحیح است، اما کدی که برای پیاده‌سازی آن نوشته شده است تمیز نیست و نیاز به تغییر دارد.

پرونده[5]های مربوط به این تمرین را می‌توانید از بخش مربوط به این تمرین در صفحه‌ی درس دریافت کنید. صورت مساله، راه حلی که برای آن ارائه شده، ورودی‌های آزمون آن و یک داور خودکار[6] برای بررسی ورودی‌های آزمون به شما داده شده است.

معیارهای کد تمیز

عواملی در کد وجود دارند که ممکن است باعث کثیف شدن آن شوند. در ادامه برخی از این عوامل توضیح داده شده اند. توجه کنید که معیار نمره دهی در این تمرین همین عوامل خواهد بود و به ازای هر یک از موارد زیر که در کد شما وجود داشته باشد، مقداری از نمره‌ی شما کاسته خواهد شد. ساختار کلی و طراحی پیشین کدی که در اختیارتان قرار داده شده، نباید تغییر کند و فقط ساختار درونی کد که شامل مواردی است که در ادامه آمده می‌تواند تغییر کند. همانطور که گفته شد، تغییرات را مرحله به مرحله انجام داده و پس از اجرای هر مرحله، با اجرای موارد آزمون اطمینان پیدا کنید که عملکرد برنامه با مشکل مواجه نشده باشد. دقت داشته باشید که در فرایند تمیز کردن کد، نباید کارایی برنامه از دست برود و در انتها نیز باید آزمون‌ها به درستی اجرا شوند.

معیارهای زیر، خلاصه‌ای از کتاب Clean Code هستند. عبارت مقابل هر بخش شماره‌ی فصل مرتبط با آن بخش را در کتاب نشان می‌دهد.

1 نام‌گذاری

1.1 نام‌گذاری متغیرها

- استفاده از نام‌های نامرتبط کار درستی نیست. مثلا استفاده از متغیرهایی با نام‌های a و b که هیچ توضیحی در مورد کارکرد متغیر ارائه نمی‌دهند و خواننده را گیج می‌کنند. (فصل 17، N1)

[3] struct
[4] functionality
[5] file
[6] automatic judge

- نام متغیر باید کاربرد و مکان استفاده از متغیر را نشان دهد. اسامی کلاس‌ها[7]، ساختارها[8] و اشیا[9] باید به صورت عبارت‌های اسمی[10] باشد. مثلا: customer, account

1.2 نام گذاری توابع

- نام تابع باید وظیفه‌ی تابع و تاثیرات جانبی[11] احتمالی تابع بر محیط را توضیح دهد. اسامی توابع باید عبارت‌های امری باشند و با حرف کوچک شروع شوند؛ مثلا: removeUser, get_flagged_cells.

  علاوه بر این، اسامی توابع را می‌توان به صورت camelCase یا با استفاده از خط زیرین[12] نوشت.

  در حالت camelCase اولین حرف اولین کلمه در نام تابع با حرف کوچک و اولین حرف باقی کلمات با حرف بزرگ نمایش داده می‌شود؛ مانند: removeUserData.

  در حالت استفاده از خط زیرین، بین همه‌ی کلمات در نام تابع یک حرف "_" (خط زیرین) قرار می‌گیرد؛ مانند: get_user_info.

  در زبان‌های C, C++, Python معمولا از snake_case، و در زبان‌های Java و JavaScript از camelCase استفاده می‌شود.

2 توابع

- توابع باید تا حد امکان کوتاه باشند. طول توابع به ندرت باید به ۲۰ خط برسد.

- هر تابع باید حداکثر به یک سطح پایین‌تر دسترسی داشته باشد؛ مثلا حرکت با یک حلقه روی لیستی از اشیا و تغییر ویژگی[13]های هر کدام از اشیا، دسترسی تابع به دو سطح پایین‌تر محسوب می‌شود؛ این عملیات باید در تابعی جداگانه پیاده‌سازی شود. عموما پیچیدگی تابع و نوع دسترسی های آن باید به گونه ای باشند که کد مربوط به آن بیشتر از 2-3 دندانه[14] داخل نشده باشد.

- تعداد آرگومان‌های تابع تا حد امکان کم (ترجیحاً ۱ یا ۲ و حداکثر ۳ تا) باشد. گاهی می‌توان از آرگومان‌هایی از نوع اشیا یا ساختارها برای بسته‌بندی[15] چند آرگومان مرتبط و کاهش تعداد آرگومان‌های توابع استفاده کرد؛ مثلا به جای دو متغیر از نوع double از یک ساختار از نوع Point استفاده کنیم.

- انجام بیش از یک کار در یک تابع درست نیست. هر تابع باید فقط یک کار را انجام دهد و این کار را به شیوه‌ی درستی پیاده و اجرا کند. در واقع توابع باید یک مجموعه از وظایف که هدف واحدی را دنبال می‌کنند و در یک سطح از انتزاع هستند را اجرا کنند. این ایراد ممکن است خود را در نام‌گذاری تابع نشان دهد. اگر برای نام گذاری

[7] classes
[8] structures
[9] objects
[10] noun phrases
[11] side-effects
[12] underscore
[13] property
[14] indent
[15] wrapping

تابع خود مجبور شدید از دو فعل استفاده کنید (مثلا calculate_and_show) یعنی تابع بیشتر از یک کار انجام می‌دهد و باید تقسیم شود (مثلا به دو بخش calculate و show). (فصل 17، G30)

- به عنوان یک راهنما، توصیه می‌شود تا حد امکان تابع‌ها را به توابع کوچکتر تقسیم کنید. این کار را تا جایی پیش ببرید که احساس کنید استخراج یک تابع از تابع فعلی، نام و مفهوم تابع فعلی را تکرار کند.

- استفاده از پرچم[16]ها (معمولا آرگومان از نوع بولی[17]) برای تعیین نحوه‌ی عملکرد تابع کار درستی نیست. مثالی از این کار ارسال یک متغیر به نام flag به تابع، فقط برای اجرای یک بخش کد در حالتی خاص است. چنین تابعی در واقع حاصل ادغام دو تابع مختلف است که باید به صورت جدا از هم پیاده‌سازی شوند و در زمان مناسب فراخوانی شوند(فصل 17، F3). یک نمونه‌کد از انجام این اشتباه به صورت زیر است:

```
int add_customer(string name, string password,
  bool passIsValid) {
  if(passIsValid) {
    shop.add(customer, password);
    print_message("Customer added successfully!");
    return 1;
  } else {
    print_message("Please enter another password!");
    return 0;
  }
}
```

که می‌توان آن را به صورت زیر اصلاح کرد:

```
void add_customer(string name, string password) {
  shop.add(customer, password);
  print_message("Customer added successfully!");
}

int register(string name, string password) {
  if(is_valid(password)) {
    add_customer(name);
    return 1;
  } else {
    print_message("Please enter another password!");
    return 0;
  }
}
```

[16] flag
[17] boolean

3 قالب‌بندی

3.1 دندانه‌گذاری[18]

دندانه‌گذاری در کد اهمیت بالایی دارد و حتما هر محدوده[19] باید یک دندانه داخل‌تر باشد. هم‌چنین هر تابع باید حداکثر یک یا دو دندانه داخل رفته باشد.

3.2 سازگاری[20]

سازگاری یکی دیگر از نکات مهم در نوشتن کد است. سعی کنید که همیشه از یک الگو و روند در پیاده‌سازی و نام‌گذاری‌های خود استفاده کنید. (فصل ۱۷، G11)

- در نام‌گذاری توابع و متغیرها باید از یک روش واحد نام‌گذاری استفاده شده باشد؛ مثلا یا همه‌ی متغیرها به صورت camelCase یا همه به شکل snake_case نام‌گذاری شده باشند. این موارد شامل اسم کلاس‌ها (که باید به صورت PascalCase باشند) نمی‌شود. در هر صورت، دیگر قوانین نام‌گذاری نیز باید رعایت شوند.

- نوع دندانه‌گذاری، طول دندانه‌ها و نحوه‌ی استفاده از فضای سفید[21] در طول کد باید ثابت باشد. این قانون شامل فاصله‌های بین عملگرها و عملوندها و حتی اجزای کوچکی مانندِ ";" پایان هر دستور نیز می‌شود. به عنوان مثال دو قطعه کد زیر نمونه یک قالب‌بندی ناسازگار و تصحیح شده‌ی آن است. در کد ناسازگار، هیچ کدام از قوانین سازگاری رعایت نشده است. دقت کنید که حتما لازم نیست نحوه‌ی فاصله‌گذاری‌ها و قرار گرفتن کروشه‌ها و... مانند نمونه کد سازگار باشد. مهم این است که یک سبک از کدنویسی در کل طول کد رعایت شده باشد.

کد ناسازگار

```
if(language == English)
{
  cout << "Hello\n";
  add_american_customer(name);
}
if(language==Farsi) {
  cout<< "Salam\n" ;
  addPersianCustomer(name);
}
```

[18] indentation
[19] scope
[20] consistency
[21] whitespace

کد سازگار

```
if(language == English) {
  cout << "Hello\n";
  add_american_customer(name);
}
if(language == Farsi) {
  cout << "Salam\n";
  add_persian_customer(name);
}
```

4 مشکلات دیگر

اشکالات دیگری نیز که ممکن است در کد شما دیده شود که باید آنها را برطرف کنید عبارت‌اند از:

- **کد تکراری**[22]: از مهم‌ترین نکاتی که باید در این تمرین رعایت کنید، جلوگیری از تکرار کد است. (G5) اگر قطعه‌ای از کد هست که کار خاصی انجام می‌دهد و چند بار در طول برنامه تکرار شده است، آن را به صورت یک تابع در بیاورید.
- **کدهای مرده**[23]: کدهایی که در هیچ مسیری از اجرای برنامه اجرا نمی‌شوند، نباید در متن برنامه وجود داشته باشند. (G9)
- استفاده از **اعداد جادویی**[24]: اعداد و ثابت‌ها نباید به طور مستقیم در کد استفاده شوند؛ بلکه باید در ثابت[25]ها ذخیره شوند و از این متغیرها در کد استفاده شود؛ مثلا عدد π را باید در ابتدای برنامه در ثابتی به نام PI ذخیره کنیم و از این ثابت در بقیه کد استفاده کنیم (به نمونه‌ی زیر دقت کنید). (G25)

```
#include <...>

const double PI = 3.14;

int main() {...}
```

[22] duplication
[23] dead codes
[24] magic numbers
[25] constants

خلاصه‌ها

سه نمونه خلاصه‌ی قابل استفاده را در لینک‌های زیر می‌توانید مشاهده کنید:

- Clean Code
- Refactoring
- Google C++ Style Guide

معرفی git

گیت یک سیستم کنترل نسخه است که بخش زیادی از توسعه‌دهندگان سراسر دنیا از آن استفاده می‌کنند. در روند انجام یک پروژه در طی زمان، ممکن است نیاز پیدا کنید که تغییرات مختلفی که روی کد انجام داده‌اید را ردیابی و به گونه‌ای ذخیره کنید. در واقع دوست دارید بدانید که چه تغییری، توسط چه کسی و در چه زمانی صورت گرفته است. نیاز به چنین ابزاری زمانی مبرم می‌شود که در پروژه bug یا خطایی پیدا شود.

گیت در واقع برای شما این امکان را فراهم می‌کند که بتوانید تمام تغییرات ذکر شده در بالا را رصد کنید. علاوه بر این، امکان توسعه هم‌زمان پروژه توسط چند نفر را فراهم می‌کند. در واقع شما با ایجاد **شاخه**‌های مختلف برای پروژه، این امکان را فراهم می‌کنید که چند شخص یا تیم بر روی قسمت‌های مختلف پروژه کار کنند بدون اینکه تغییراتی که می‌دهند تیم‌های دیگر را تحت تاثیر قرار دهد. در ادامه یک روند ساده مراحلی که برای یک تغییر در گیت طی می‌شود آورده شده است.

برای ایجاد یک پروژه گیت، در صورتی که پروژه قبلاً در مخزنی برپا شده باشد، عموماً از دستور

```
git clone <repo> <directory>
```

استفاده می‌شود. با استفاده از این دستور، پروژه‌ای با محتوای مخزن به آدرس repo در پوشه‌ی directory ساخته می‌شود.

در صورتی که پروژه در مخزنی برپا نشده باشد، با استفاده از دستور

```
git init <directory>
```

می‌توان پروژه گیت جدیدی در پوشه directory ایجاد کرد.

پس از ایجاد یا کلون کردن یک مخزن گیت، می‌توانیم به کمک دستور زیر مخزن را پیکربندی[26] کنیم:

```
git config --[local | global] <name> <value>
```

معمولا مقداردهی به دو متغیر نام و ایمیل کاربر در یک مخزن، برای ثبت کردن تغییرات در آن مخزن لازم است.

اگر پوشه‌ای که پروژه در آن وجود دارد یک مخزن[27] گیت باشد، گیت تمام تغییراتی که در هر فایل پروژه وجود دارد را زیر نظر می‌گیرد. این تغییرات با استفاده از دستور

```
git status
```

[26] configure

[27] repository

قابل مشاهده است. همچنین با استفاده از دستور

```
git add <file or directory>
```

می‌توان تغییرات ایجاد شده در فایل‌های مختلف را به staging area منتقل کرد تا در قالب بسته‌هایی به عنوان تغییر ثبت شوند. به این بسته‌ها commit گفته می‌شود. در واقع کامیت‌ها کوچک‌ترین واحدهای تغییر در گیت هستند. هر کامیت بهتر است که یک تغییر معنادار بر روی پروژه اعمال کند.

هر کامیت به همراه یک پیام ثبت می‌شود که خلاصه‌ای از شرح آن تغییر است که در این کامیت اعمال شده است. شما می‌توانید با دستور

```
git commit -m <message>
```

تغییراتی که در staging area وجود دارد را به شکل یک کامیت و با یک پیام خاص، ثبت کنید.

همچنین می‌توانید از دستورات checkout برای ساختن و جابه‌جا شدن بین شاخه‌های مختلف، stash برای بررسی و چک کردن تغییرات و ذخیره‌سازی موقت، rebase برای تغییر دادن شاخه اصلی و merge برای یکسان‌سازی دو شاخه استفاده کنید. همچنین ممکن است در طول تمرین، با ناسازگاری[28] دو شاخه برخورد کنید که می‌توانید از منابع معرفی شده برای برطرف‌سازی آن‌ها استفاده کنید.

معرفی Github

تا اینجا تمام توضیحات داده شده مربوط به ابزار گیت بود. در بسیاری از مواقع و به علت‌های گوناگون دوست داریم که مخزن گیت ما نه تنها بر روی کامپیوتر خودمان، بلکه از طریق اینترنت نیز قابل دسترسی باشد. بدین شکل توسعه دهندگان دیگر در نقاط دیگر دنیا می‌توانند با این پروژه تعامل داشته باشند (این سنگ بنای پروژه‌های متن‌باز[29] نیز هست). گیت‌هاب یک وب‌سایت میزبانی از گیت است. در واقع گیت‌هاب به شما این امکان را می‌دهد که پروژه‌هایی را که به صورت محلی[30] در سیستم خود دارید را روی فضای ابری گیت‌هاب به صورت دوردست[31] در اختیار دیگران قرار دهید. همچنین با استفاده از دستورات pull و push می‌توانید با مخزن‌های دوردست گیت کار کنید.

```
git pull # pulls the changes from remote on the branch
git push # pushes the committed changes to the branch
```

منابع بیشتر برای یادگیری گیت

برای آشنایی بیش‌تر با گیت، می‌توانید از منابع زیر استفاده کنید:

- ویدیوی آموزشی گام‌به‌گام راه‌اندازی و کار کردن با گیت که توسط یکی از دستیاران آموزشی در ترم‌های گذشته تهیه شده است.
- راهنمای Atlassian

[28] conflict
[29] open-source
[30] local
[31] remote

- راهنمای freecodecamp
- راهنمای w3schools
- راهنمای شاخه‌بندی
- راهنمای برخورد با ناسازگاری(۱)
- راهنمای برخورد با ناسازگاری(۲)

نحوهٔ تحویل

- با توجه به اینکه در ادامه‌ی ترم، یک الی دو تمرین را می‌توانید به صورت گروه‌های دو نفره تحویل دهید، می‌توانید برای شروع یادگیری کار تیمی، این تمرین را نیز به صورت گروهی انجام دهید. گروهی انجام دادن این تمرین، الزامی نیست و می‌توانید به صورت تکی نیز آن را تحویل دهید.

- توجه داشته باشید که در صورت تمایل، می‌توانید گروه خود را در طی تمرین‌ها تغییر دهید.

- در پرونده فشرده‌شده‌ای که در ای‌لرن آپلود شده است، چهار کد C0.cpp تا C3.cpp موجود است. رقم عددی کدی که شما باید روی آن بازآرایی را انجام دهید، باقی‌مانده یکان شماره دانشجویی شما (مجموع یکان شماره دانشجویی‌های شما در صورتی که به صورت گروهی تمرین را انجام می‌دهید) به ۴ است. برای مثال اگر شماره دانشجویی شما ۸۱۰۱۰۱۰۰۰ باشد، کد شما C0.cpp خواهد بود.

- پس از دریافت پرونده‌های تمرین، یک مخزن گیت در پوشه‌ای که به تمرین اختصاص داده‌اید، بسازید. یک مخزن گیت‌هاب نیز با نام ABonus-SID بسازید که SID شماره‌ی دانشجویی شماست (برای مثال اگر شماره‌ی دانشجویی شما ۸۱۰۱۰۱۹۹۹ است، نام این مخزن را ABonus-810101999 بگذارید. همچنین در صورتی که به صورت گروهی این تمرین را تحویل می‌دهید و شماره دانشجویی اعضای تیم، ۸۱۰۱۰۱۹۹۸ و ۸۱۰۱۰۱۹۹۹ باشد، نام این مخزن را ABonus-810101998-810101999 بگذارید). مطابق توضیحات ارائه شده، بازآرایی را مرحله به مرحله انجام دهید و روی مخزن گیت‌هاب push کنید. سپس آدرس مخزن گیت‌هاب خود را به همراه شناسه‌ی آخرین کامیت و شناسه(های) گیت‌هاب خودتان در محل بارگذاری تمرین در ای‌لرن بنویسید.

- مشارکت شما در این تمرین، از روی کامیت‌های ثبت شده در مخزن گیت‌هاب بررسی می‌شود.

- مطمئن باشید که دسترسی مخزن گیت‌هاب خود را در حالت **شخصی**[32] قرار می‌دهید. کاربر AP-S02 را نیز به مخزن گیت‌هاب خود اضافه کنید.

- پیام‌های کامیت خود را واضح بنویسید و تغییرات هر کامیت را به طور خلاصه در پیام آن بیاورید. از این لینک می‌توانید به عنوان راهنما استفاده کنید. توجه کنید که بخش قابل توجهی از ارزیابی این تمرین، مربوط به پیام‌های کامیت شما و تناسب آن‌ها با تغییرات ایجاد شده است.

[32] private

- به کمک ورودی‌های آزمون، از درستی کد خود پس از هر مرحله‌ی بازآرایی مطمئن شوید. همچنین برای آشنایی بیشتر با شل اسکریپت[33]، می‌توانید از داوری[34] که در اختیار شما قرار داده شده است نیز استفاده کنید. این مهارت می‌تواند در ادامه تحصیل شما در حوزه مهندسی نرم‌افزار تاثیرگذار باشد. برای آشنایی بیشتر نیز می‌توانید از این لینک و این لینک استفاده کنید.

- هدف این تمرین یادگیری شماست. لطفاً تمرین را خودتان انجام دهید. در صورت کشف تقلب مطابق قوانین درس با آن برخورد خواهد شد.

[33] shell script

[34] judge