دانشکده مهندسی
کامپیوتر و فناوری اطلاعات

۱۳۹۹/۸/۱۸

تمرین سوم

مهندسی نرم افزار ۲

گروه {۲}

اعضاء گروه:

۱) محمدرضا اخگری زیری - ۹۶۳۱۰۰۱

۲) محمدعلی کشاورز - ۹۶۳۱۰۶۱

۳) علی نظری - ۹۶۳۱۰۷۵

۱) معماری مبتنی بر مولفه[1] و معماری سرویس گرا[2] چه تفاوت هایی دارند؟

پاسخ:

در ابتدا تعریف مختصری از این دو معماری ارائه می‌کنیم.

- **معماری مبتنی بر مولفه:**

مولفه یک شی نرم‌افزاری است؛ به منظور تعامل با سایر مولفه‌ها، در هر کامپوننت فانکشنالیتی‌ها و مشخصه‌های خاصی وجود دارد که معنای واحد و یکپارچه‌ای دارد. یک مولفه به طور واضح دارای رابط کاربری مشخص شده و مشترک برای همه است. تمرکز اصلی این نحوه از معماری روی تجزیه سیستم به مولفه‌های معنادار است.

یکی از اصلی ترین اهداف این پارادایم، کم کردن تایم تو مارکت و افزایش پروداکتیویتی است. در این روش تلاش می‌شود که به جای اختراع دوباره چرخ از کامپوننت‌های موجود و استاندارد استفاده شود.

اصول اصلی معماری مبتنی بر مولفه:

- Reusability: کامپوننت‌ها معمولا به شکلی طراحی می‌شوند که بتوان آن‌ها را مجددا در کارکردهای متفاوت و نرم افزارهای متفاوت به کار برد.
- Replaceable: همه کامپوننت‌ها باید این امکان را داشته باشند که به راحتی با کامپوننت مشابه عوض شوند (با استفاده از اینترفیس ها بتوان پیاده سازی را تغییر داد).
- Not context specific: تا حد امکان نبایستی صرفا برای یک محیط و موضوع خاص طراحی شده باشند.
- Extensible: این قابلیت را دارند که بتوان به آن‌ها رفتارهای جدید اضافه کرد.
- Encapsulated: این امکان وجود دارد که بقیه بتوانند فانکشنالیتی‌ها را استفاده کنند، اما نمی‌توانند جزئیات مربوط به چگونگی پیاده سازی آن‌ها را ببینند.
- Independent: به نحوی طراحی می‌شوند که تا جای ممکن کمترین وابستگی را به سایر مولفه‌ها داشته باشند.

- **معماری سرویس گرا:**

این معماری مبتنی بر نحوه برخورد بین سرویس هاست؛ مثلا سرویس بازیابی صورتحساب‌های بانکی و... که از طریق API با سرویس ما ارتباط دارد.

[1] Component-Based Architecture
[2] Service-Oriented Architecture

سرویس مطابق با بسیاری از تعاریف شامل ۴ ویژگی است:

- از لحاظ منطقی به تنهایی، دارای یک فعالیت تجاری با نتیجه مشخص است.
- برای مصرف کنندگان یک بلک باکس است، به این معنی که مصرف کننده اطلاعی از نحوه عملکرد داخلی سیستم ندارد.
- ممکن است خودش شامل سرویس‌هایی باشد (یعنی خودش هم از سرویس های پایین دستی استفاده کند).
- جامع و کامل است و به تنهایی نیازی به سرویس‌های خارجی ندارد.

- **مقایسه این دو معماری:**

سرویس گرا	مبتنی بر مولفه
سرویس‌های دیگری که سرویس‌های مارا فراخوانی می‌کنند ثابت نیستند و به صورت پیش‌فرض مداوم تغییر می‌کنند.	کامپوننت‌ها عمدتا برای استفاده داخلی و ثابت طراحی می‌شوند. یعنی کلاس‌ها، کامپوننت‌های ثابت دیگری آن‌ها را صدا می‌زنند.
تلاشی مبنی بر کم‌کردن ارتباط بین سرویس‌ها (به صورت جنرال) صورت نمی‌پذیرد بلکه معماری اساسا روی همین ارتباط بنا شده است.	رابط‌های کاربری به طور کلی به نحوی است که خیلی سخت گیرانه روی آن‌ها نظارت می‌شود و تلاش برای کم کردن رابطه بین کامپوننت هاست.
دسترسی در قالب استانداردهای تعریف شده مانند HTML/SOAP است.	در این نحوه معماری دسترسی‌ها در حافظه اصلی است.
دسترسی از همه جا امکان پذیر است.	دسترسی در سطح موتورهای کد میانی مانند JVM و... است.
باتل‌نک در ارتباط بین سرویس‌ها رخ می‌دهد و پرفورمنس ضعیف است؛ ضمنا در بسیاری از موارد پرفورمنس وابسته به سرویس خارجی است.	پرفورمنس به واسطه ارتباط‌ها خیلی کم نمی‌شود و در این زمینه به نسبت معماری مبتنی بر سرویس پرفورمنس بهتری را ارائه می‌دهد.
برآورده کردن نیازمندی‌های غیرعملکردی‌ای نظیر امنیت و سرعت و قابل اطمینان بودن سخت‌تر است.	برآورده کردن نیازمندی‌های غیرعملکردی‌ای نظیر امنیت و سرعت و قابل اطمینان بودن سخت‌تر است.
هزینه اینترنت دارد	هزینه اینترنت ندارد

منابع استفاده شده برای این سوال:

https://sites.google.com/site/nextthoughtlabs/engineering/component-based-architecture-vs-service-oriented-architecture-part-1
http://www.petritsch.co.at/download/SOA_vs_component_based.pdf
https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm#:~:text=Component%2Dbased%20architecture%20focuses%20on,methods%2C%20events%2C%20and%20properties

۲) در مورد "معماری تمیز[3]" که توسط Robert C. Martin مطرح شده است، مطالعه کنید.[4]

الف) مهم ترین اصول و قواعد این معماری را معرفی کنید.

پاسخ:

معماری تمیز با اصول کد تمیز[5] شروع می‌شود. کلاس‌ها باید تمیز باشند تا مولفه‌ها[6] تمیز شوند و مولفه‌ها باید تمیز باشند تا سیستم تمیز باشد.

پس طبق جمله اول، خواهیم داشت:

کد تمیز از اصول پنج‌گانه SOLID پیروی می‌کند.

- Single responsibility principle

این قانون که به طور خلاصه SRP نیز نامیده می‌شود، حاکی از آن است که یک کلاس باید صرفاً یک وظیفه بیشتر نداشته باشد که در این صورت، کلاس‌ها فقط و فقط به خاطر ایجاد تغییر در وظیفه‌ای که انجام می‌دهند دستخوش تغییر خواهند شد نه چیز دیگر! کلاس‌ها می‌توانند ویژگی‌های مختلفی داشته باشند اما تمامی آن‌ها باید مربوط به یک حوزه بوده و **مرتبط** به هم باشند که در نهایت با محترم شمردن چنین قانونی، برنامه‌نویسان دیگر قادر نخواهند بود تا کلاس‌های اصطلاحاً همه‌فن‌حریف بنویسند.

- Open-closed principle

هر کلاسی باید برای توسعه یافتن قابلیت‌هایش اصطلاحاً Open بوده و دست برنامه‌نویس برای افزودن ویژگی‌های جدید به آن باز باشد اما اگر وی خواست تا تغییری در کلاس ایجاد کند، چنین امکان باید Closed بوده و او اجازه‌ی چنین کاری را نداشته باشد. فرض کنیم نرم‌افزاری نوشته‌ایم که دارای چندین کلاس مختلف است و نیازهای اپلیکیشن‌مان را مرتفع می‌سازند اما به جایی رسیده‌ایم که نیاز داریم قابلیت‌های جدید به برنامه‌ی خود بیفزاییم. بر اساس این قانون، دست‌مان برای تغییر یا بهتر بگوییم افزودن فیچرهای جدید به کلاس مد نظر باز است در حالی که این قابلیت‌های جدید باید در قالب افزودن کدهای جدید صورت پذیرد نه ریفکتور کردن و تغییر کدهای قبلی!

- Liskov's substitution principle

این اصل حاکی از آن است که کلاس‌های فرزند باید آن‌قدر کامل و جامع از کلاس والد خود ارث‌بری کرده باشند که به سادگی بتوان همان رفتاری که با کلاس والد می‌کنیم را با کلاس‌های فرزند نیز داشته باشیم به طوری که

---

[3] Clean Architecture

[4] برای مطالعه بیشتر: https://cutt.ly/lgJsQTE و https://blog.cleancoder.com

[5] Clean Code

[6] Components

اگر در شرایطی قرار گرفتید که با خود گفتید کلاس فرزند می‌تواند تمامی کارهای کلاس والدش را انجام دهد به جزء برخی موارد خاص، اینجا است که این اصل از SOLID را نقض کرده‌اید.

- Interface segregation principle

اینترفیس‌ها فقط مشخص می‌کنند که یک کلاس از چه متدهایی **حتماً** باید برخوردار باشد. در همین راستا و بر اساس این قانون، چندین اینترفیس تک‌منظوره به مراتب بهتر است از یک اینترفیس چندمنظوره است

- Dependency inversion principle

اصل DIP از دو جمله تشکیل شده است، جمله اول می‌گوید که ماژول‌های سطح بالا نباید به ماژول‌های سطح پائین وابسته باشند، بلکه هر دو آن‌ها باید به Abstractionها وابسته باشند. جمله دوم می‌گوید که Abstraction ها نباید به Detailها وابسته باشند، بلکه Detailها باید به Abstractionها وابسته باشند.

در زمینه ی مولفه‌ها نیز اصولی مطرح کرده است.

- Reuse/release equivalence principle (RRP)

کلاس‌ها و ماژول‌ها (یعنی یک جز مولفه) که با هم reuse شدند باید با هم release شوند. آنها باید شماره نسخه یکسانی داشته باشند و باید مستندات مناسب مانند تغییرات وجود داشته باشد.

- Common closure principle (CCP)

کلاس‌هایی که با هم تغییر می‌کنند باید با هم گروه شوند و بالعکس. اصل مسئولیت واحد در سطح مولفه‌ها.

- Common reuse principle (CRP)

کاربران یک مولفه را مجبور نکنید به چیزهایی که نیازی ندارند بستگی داشته باشند. اصل تفکیک رابط در سطح جز مولفه.

مجموعه اصول بعدی مربوط به اتصال مولفه‌ها[7]ست:

- Acyclic dependencies principle

هیچ دوری در نمودار وابستگی وجود ندارد. برای شکستن چرخه ها از اصل وارونگی وابستگی استفاده کنید.

- The stable dependency principle

اجزای کم ثبات باید به اجزای پایدارتر وابسته باشند.

[7] Component coupling

- Stable abstractions principle

اجزای پایدار باید انتزاعی باشند و بالعکس. (دقیقا طریقی که entity پیاده می‌شود) مثالی از یک مولفه پایدار انتزاعی، سیاست سطح بالایی است که با پیروی از اصل open-closed تغییر می‌کند.

برای معماری نیز اصولی مطرح کرده است که از روی نمودار نیز قابل فهم است:

- Testable
- Independent of frameworks
- Independent of the UI
- Independent of the database
- Independent of any external agency

متن خارجی این ویژگی‌ها معنا را می‌رساند.

ب) کاربرد اصل Dependency Inversion (از اصول SOLID) در این معماری چیست؟

پاسخ:

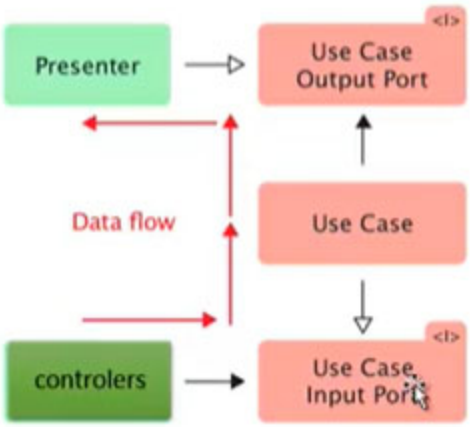
همانطورکه در قسمت قبل گفتیم اصل Dependency Inversion از دو جمله تشکیل شده است:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions

ما همچنین می‌دانیم که در معماری تمیز، جهت دسترسی‌ها از بیرون به داخل است، این بدین معناست که اجزای low level به اجزای high level وابسته هستند. برای مثال اگر ما در آینده قرار باشد نوع db خود را عوض کنیم لازم نیست تا به کد domain خود دست بزنیم.

در حقیقت در این معماری ما به جزییات توجه نمی‌کنیم (مثلاً استفاده از کدام framework) و جزییات به معماری ما وابسته می‌شوند، نه برعکس.

برای جمله‌ی دوم هم می‌توان برای مثال رابطه لایه presenter و use case را مشاهده کرد. این دو لایه با استفاده از abstraction به هم متصل خواهند شد.



عکس فوق برای این معماری در اندروید است و در نسخه‌های قدیمی این معماری است، در نسخه‌های جدید، controller پایین نیز، presenter نوشته می‌شود. همانطور که می‌بینید کلاس‌ها با استفاده از interface با همدیگر صحبت می‌کنند.

ج) آیا "معماری تمیز" در معماری مبتنی بر microserviceها نیز کاربرد دارد؟ مختصرا بحث کنید.

پاسخ:

طبق گفته های عمو باب در وبلاگش:

هیچ چیزی در Clean Architecture ایجاب نمی‌کند که پیام‌ها همزمان یا ناهمزمان باشند. هیچ چیز مانع از ارسال آن پیام‌های درخواست و پاسخ به سرور دیگری نمی‌شود. هیچ چیزی در مورد معماری مانع از اجرایی شدن اجزای کوچک برای برقراری ارتباط با HTTP از طریق REST نیست.

بنابراین ، یک معماری microservice می تواند مطابق با معماری تمیز باشد. در واقع ، اگر می خواستم سیستمی با استفاده از micro-services ایجاد کنم ، مطمئناً این مسیر را دنبال خواهم کرد.

یک موضوع مقیاس‌پذیری مولفه‌هاست:

برای استقرار مولفه‌ها راه‌های مختلفی است (اگر بخواهید این راه ها به ترتیب scalability در منبع آمده است)، معماری خوب به نحوه ی استقرار اجزا کاری ندارد و درباره ی آنها چیزی نمی‌داند.

از گفته های عمو باب ، "microservices یک روش برای استقرار است ، نه یک معماری[8]"، هر microservice باید قابل استفاده و نگهداری توسط تیم‌های مختلف (که می‌تواند در مکان‌های مختلف جغرافیایی باشد) باشد. هر تیم می‌تواند معماری، زبان برنامه نویسی، ابزار، فریم ورک و غیره خود را انتخاب کند و مجبور کردن هر تیم به استفاده از زبان برنامه‌نویسی منفرد/یکسان یا ابزار یا معماری، خوب به نظر نمی‌رسد. بنابراین هر تیم microservice باید بتواند معماری خود را انتخاب کند.

چگونه هر تیم می تواند microservice خود را رمزگذاری کند/ نگهداری/ استقرار دهد بدون اینکه با کد تیم های دیگر مغایرت داشته باشد؟ این سوال ما را به چگونگی تفکیک microservice می رساند. باید براساس ویژگی تفکیک شود.

پس از جدا کردن microservice، ارتباط بین آنها جزئیات پیاده سازی است. این کار می تواند از طریق REST API / web-socket و غیره انجام شود. در داخل هر سرویس کوچک، اگر تیم تصمیم به دنبال Clean Architecture بگیرد، آن‌ها می‌توانند چندین لایه بر اساس اصول Clean Arch داشته باشند.

همچنین میتوان از DDD برای تقسیم بندی استفاده کرد. در اکثر موارد استفاده از DDD راه مطمئن برای تقسیم بندی سرویس هاست.

منابع:

https://blog.cleancoder.com/uncle-bob/2014/10/01/CleanMicroserviceArchitecture.html

[8] Micro-services are deployment option, not an architecture.

۳) اصول SOLID و YAGNI را در نظر بگیرید:

الف) آن ها را با یکدیگر مقایسه کنید.

پاسخ:

پنج اصل ذکر شده در SOLID که بیشتر در برنامه‌نویسی شی‌گرا دیده می‌شوند (البته مفاهیمش به صورت کلی در سایر پارادایم‌های برنامه‌نویسی نیز قابل استفاده است)، بر این تمرکز دارند که با کمک آن‌ها بتوانیم نرم‌افزاری تولید کنیم که در آینده بتوانیم راحت‌تر آن را مقیاس‌پذیر کنیم و در فرایند توسعه‌ی چابک آن‌ها را همواره بهبود ببخشیم. همچنین به کمک آن‌ها می‌توانیم ساده‌تر با تغییراتی که در توسعه‌ی پروژه نیاز می‌شود، کنار بیاییم و آن‌ها را در پروژه اعمال کنیم زیرا از قبل به شکلی کدها را زده‌ایم که اکنون کار ساده‌تری داریم و در نهایت به ما کمک می‌کنند تا کیفیت کدی که می‌زنیم را بهبود ببخشیم.

اما از طرف دیگر YAGNI بر این اصل تاکید دارد که سعی کنیم ساده‌تر نگاه کنیم و به این توجه کنیم که چیزی را توسعه دهیم که به آن احتیاج داریم و چیزهایی که نیاز نداریم را رها کنیم چون ممکن است به چیزهایی فکر کنیم که هیچ وقت به آن‌ها احتیاج پیدا نمی‌کنیم.

ب) آیا این دو رویکرد با یکدیگر در تناقض هستند؟

پاسخ:

به ظاهر این طور است که یکی می‌گوید فقط به چیزی که الان نیاز داریم فکر کنیم و اصل دیگر می‌گوید که این اصول را رعایت کنیم تا علاوه بر افزایش کیفیت کد، در آینده نیز کار ساده‌تری داشته باشیم. اما به نظر ما این طور نیست که این دو اصل با هم در تناقض باشند چون افزایش کیفیت کد تناقضی ندارد با اینکه ما سعی کنیم ساده‌تر به مسائل نگاه کنیم و خود را درگیر مسائلی که الان به آن‌ها نیازی نداریم، نکنیم. به بیان دیگر ما می‌توانیم یک مجموعه با معماری صحیح داشته باشیم ولی به صورت همزمان به مسائلی که فقط در حاضر به آن‌ها نیاز داریم بپردازیم و خود را درگیر چیزهایی که نیاز نداریم، نکنیم.

ما باید با توجه به هر دوی آن‌ها توسعه‌ی خود را انجام دهیم و با در نظر گرفتن شرایط و نیازهایمان اصول مناسب‌تر را رعایت کنیم.

ج) تجربه یا توصیه خود در استفاده از این دو رویکرد را مختصرا شرح دهید.

پاسخ:

با توجه به شرایط و کارهایی که ما در انجام آن‌ها مشارکت داشتیم، مخصوصا در زمان‌هایی که به صورت Pair داشتیم کدی را می‌زدیم، یکی از افراد پیشنهاد می‌داد که مثلا در پیاده‌سازی یک بخش به این توجه کنیم که ممکن است آن را بخواهیم به صورت دیگری نیز استفاده کنیم و نفر دیگر با اشاره به YAGNI سعی می‌کرد که با جلوگیری از پرداختن به چیزهایی که به آن‌ها نیازی نداریم از پیچیدگی و همچنین طولانی شدن فرایند توسعه جلوگیری کند که به نظر هم کار درستی بود و در بسیاری از موارد دیگر هیچ‌وقت لازم نشده بود که به سراغ آن موارد برویم و نیاز به تغییر آن‌ها نیز نبود زیرا به همان صورت ساده که نوشته شده بودند هنوز هم کافی بودند.

۴) درباره تحلیل معماری[9] و روش های آن تحقیق کنید.[10] (امتیازی)

الف) دو روش را انتخاب کرده و مختصرا معرفی کنید. (ترجیحا با ذکر دلیل انتخاب)

پاسخ:

الف) ما روش‌های SBAR و ATAM را انتخاب می‌کنیم (البته چون در مقاله نبود روش FAAM را انتخاب نکردیم، در حالی که این روش پارامترهای بیشتری نسب به ATAM را پوشش می‌دهد). روش‌های انتخاب شده برخلاف ۴ روش اول که فقط مبتنی به سناریو هستند، تکنیک‌های دیگری را در نظر می‌گیرند (ATAM تجمیعی از سوالات موجود و تکنیک مبتنی بر مبتریک است ولی SBAR مبتنی بر سناریو، مدل‌های ریاضی، شبیه‌سازی و نتایج عینی است).

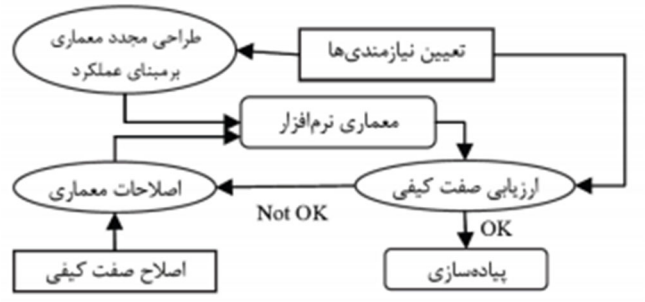
روش ATAM: روش ATAM به صورت کاملاً جدا از روش SAAM و روش‌های مبتنی بر آن، توسعه یافت و هدف از آن تحلیل صفات کیفیتی خاص بر اساس معماری بود. ATAM بر روی جنبه‌های کیفیتی معماری با جزئیات بیشتر بحث می‌کند و در عمل نسخه‌ی تقویت‌شده‌ای از SAAM می‌باشد. ATAM به عنوان یک مدل مارپیچی از طراحی در سال ۱۹۹۸ مطرح شد و در سال ۱۹۹۹ مدل مارپیچی تحلیل و طراحی که تکامل فعلی و میزان پیشرفت را شرح می‌داد، مطرح گردید.

[9] Architecture Analysis

[10] به قسمت "مطالعه بیشتر" در مودل مراجعه کنید.

ATAM یک روش مبتنی بر سناریو برای ارزیابی صفات کیفیتی مانند: قابلیت اصلاح‌پذیری، قابلیت حمل، قابلیت توسعه و قابلیت تجمیع می‌باشد. این روش چگونگی ارضاشدن اهداف کیفیتی ویژه توسط معماری نرم‌افزار را تحلیل می‌کند و یک فرآیند تکرارپذیر است.

روش SBAR: چهار تکنیک مختلف برای ارزیابی مشخه‌های کیفیتی استفاده می‌شود که عبارتند از: سناریوها، شبیه‌سازی‌ها، مدل‌سازی ریاضی و استدلال بر مبنای تجربه. برای هر مشخصه کیفیتی، تکنیک مناسب انتخاب می‌شود. یک خصوصیت برجسته‌اش این روش آن است که برای ارزیابی معماری سیستم موجود، خود سیستم می‌تواند استفاده شود. فرآیند ارزیابی می‌تواند به روش کامل یا آماری انجام شود. در روش اول، مجموعه‌ای از سناریوها تعریف و با هم ترکیب می‌شوند. دومین روش تعریف کردن مجموعه‌ای از سناریوها است که یک نمونه ساده بدون همه موارد را می‌سازد. SBAR به عنوان روشی برای اندازه‌گیری سیستم نرم‌افزاری شناخته شده است. در زیر شکل مربوط به مراحل این روش آمده است.



ب) پس از انتخاب یا تعریف یک مجموعه معیار ارزیابی، این دو روش را با یکدیگر مقایسه کنید.

پاسخ:

ب) مجموعه ارزیابی که ما انتخاب کردیم Quality attributes است. در این مجموعه دو روش را از نظر نیازمندی‌های غیرعملکردی‌ای که ارزیابی می‌کند، مقایسه می‌کند.

برای روش ATAM معیارهایی که اندازه گرفته می‌شود شامل modifiability، security، performance و availability هستند و برای روش SBAR معیارهایی که اندازه گرفته می‌شود شامل high performance، reusability و همچنین تمامی موارد ATAM هستند.

پس همانطور که گفته شد، معیارهایی که SBAR ارزیابی می‌کند از ATAM بیشتر است.

مهندسی نرم افزار ۲

دکتر طارمی‌راد

تمرین سوم

- پاسخ تمرین ها را به زبان فارسی و به صورت تایپ شده، در قالب یک فایلPdf ، در مودل بارگزاری کنید.
- سوالات خود را می‌توانید از طریق ایمیل از دستیاران تدریس بپرسید.
- فایل پاسخ تمرین را تنها با قالب **SE2-HW3-GroupNumber.pdf** در مودل بارگزاری کنید.
- بارگزاری تمرین توسط یکی از اعضاء گروه کافی است.
- برای پاسخ‌های هر قسمت منابع استفاده شده را درج نمائید.
- فایل زیپ ارسال **نکنید**.
- به ازای هر روز تاخیر در تحویل تمرین ۲۰٪ از نمره تمرین کسر خواهد شد.
- حداقل برخورد به پاسخ‌های مشابه، تخصیص نمره کامل منفی به طرفین خواهد بود.