آزمون نرم افزار

تمرین شماره 2 9 آبان 1402

محمدرضا بخشایش – 810199381

امین باهنر – 810199548

سوال 1)

Constructor injection پرکاربردترین و توصیه شده ترین نوع DI در جاوا است. این شامل انتقال وابستگی ها به عنوان پارامتر به سازنده کلاسی است که به آنها نیاز دارد. به این ترتیب، می‌توانید اطمینان حاصل کنید که کلاس همیشه در یک وضعیت معتبر، با تمام وابستگی‌های آن اولیه و آماده استفاده است.Constructor injection همچنین کد شما را خواناتر می کند، زیرا می توانید به وضوح ببینید که یک کلاس چه وابستگی هایی دارد و از کجا آمده است. علاوه بر این، Constructor injection شما را قادر می سازد از فیلدهای نهایی استفاده کنید که می تواند از تغییرات تصادفی جلوگیری کرده و عملکرد را بهبود بخشد. با این حال،Constructor injection نیز دارای معایبی است. این می تواند سازنده های شما را بیش از حد طولانی و پیچیده کند، به خصوص اگر وابستگی های زیادی یا وابستگی های تو در تو داشته باشید. همچنین می‌تواند خرد کردن کد شما را سخت‌تر کند، زیرا باید هر بار که می‌خواهید کلاسی را آزمایش کنید، اشیاء ساختگی را ایجاد کرده و به سازنده منتقل کنید. علاوه بر این، تزریق سازنده از وابستگی های اختیاری یا dynamic پشتیبانی نمی کند، زیرا نمی توانید آنها را پس از ایجاد کلاس تغییر دهید یا حذف کنید.

Setter injection یکی دیگر از انواع رایج DI در جاوا است. این شامل ارائه متدهای تنظیم کننده برای وابستگی های یک کلاس و فراخوانی آنها پس از نمونه سازی کلاس است. به این ترتیب، می توانید انعطاف و کنترل بیشتری بر روی زمان و نحوه تزریق وابستگی ها داشته باشید. Setter injection همچنین به شما امکان می دهد وابستگی های اختیاری یا dynamic داشته باشید، زیرا می توانید هر زمان که بخواهید آنها را تغییر دهید یا حذف کنید. علاوه بر این، Setter injection می‌تواند کد شما را خرد کند، زیرا می‌توانید از روش‌های setter برای تزریق اشیاء ساختگی بدون ایجاد یک نمونه جدید از کلاس استفاده کنید.

با این حال Setter injection دارای معایبی نیز می باشد. این می تواند کلاس شما را تغییرپذیر و در برابر حالت های ناسازگار آسیب پذیر کند، زیرا وابستگی ها می توانند پس از ایجاد کلاس تغییر یابند یا باطل شوند. Setter injection همچنین باعث می‌شود کد شما کمتر قابل خواندن باشد، زیرا نمی‌توانید با نگاه کردن به سازنده بفهمید که یک کلاس چه وابستگی‌هایی دارد و از کجا آمده‌اند. علاوه بر این،Setter injection از فیلدهای نهایی پشتیبانی نمی کند، که می تواند منجر به تغییرات تصادفی و مشکلات عملکرد شود.

Field injection کمترین نوع DI در جاوا است. این شامل استفاده از حاشیه نویسی، مانند

@Autowired یا @Inject، برای علامت گذاری فیلدهایی است که باید وابستگی تزریق شوند. به این ترتیب، می توانید از نوشتن سازنده یا تنظیم کننده برای وابستگی ها اجتناب کنید و اجازه دهید یک framework یا container را برای شما انجام دهد.Field injection می‌تواند کد شما را مختصر و ساده کند، زیرا فقط باید فیلدها را مشخص کنید و آنها را حاشیه‌نویسی کنید.

با این حال،Field injection نیز معایب زیادی دارد. این می تواند کلاس شما را محکم به framework یا container که تزریق را انجام می دهد متصل کند و آزمایش و استفاده مجدد آن را سخت تر کند.Field injection همچنین کلاس شما را مات و شکننده می کند، زیرا نمی توانید ببینید یا کنترل کنید که یک کلاس چه وابستگی هایی دارد و از کجا می آیند. علاوه بر این، تزریق فیلد از فیلدهای نهایی، وابستگی های اختیاری یا پویا یا اعتبار سنجی سازنده پشتیبانی نمی کند، که می تواند مشکلات و اشکالات زیادی را ایجاد کند.

سوال 2)

الف) به این دلیل که ابجکتی که دارد تست می‌شود و میخواهد از کلاس دیگری استفاده کند فکر میکند دارد با یه کلاس واقعی کار میکند در حالی که اینطور نیست و ما آن ابجکت را جعل کردیم و آنطور که خودمان میخواهیم تعیین میکنیم که متد هایش باید چه چیزی را برگرداند و در اصل داریم تقلب میکنیم.

ب)

Dummy: ساده ترین نوع test double است. چرا که تنها باید نیاز هایی که در سیستم مورد آزمایش، type hint شده است را برطرف کند.

Stub: هنگامی استفاده می شود که یک Dummy نیاز دارد مقدار خاصی را برای تست برگرداند.

Spy:این نوع از test double نحوه ی استفاده از شی را ضبط می کند و سپس این اطلاعات را در اختیار ما قرار می دهد تا بر مبنای آن تست بنویسیم. معمولا هنگامی از Spy استفاده می شود که از بیشتر متد های یک کلاس استفاده می کنیم اما نیاز است که بدانیم بعضی از متد های آن صدا زده شدند یا مقدار خاصی را به عنوان ورودی دریافت کردند. این نوع از test double ها با Mock شباهت زیادی دارند

Mock : نوع خاصی از test double اند که انتظاراتی از صدا زده شدن یا نشدن یک متد دارند. همچنین آنها می توانند درمورد ورودی ها و خروجی های یک متد نیز انتظاراتی را تعریف کنند. درصورتی که یکی از انتظارات Mock برآورده نشود یک Exception را throw می کند که باعث fail شدن تست ما می شود. یکی از تفاوت های بزرگی که هنگام تست نویسی با mock ها این است که Mock ها مانند سایر assertion ها ادعایی را اثبات نمی کنند بلکه انتظاراتی را تعیین می کنند که در صورت برآورده نشدن آن ها تست fail می شود.

Fake : یک پیاده سازی کاملا مشابه از کلاسی است که سیستم مورد آزمایش ما نیاز دارد. این کلاس هرگز در کد های واقعی ما استفاده نمی شود. از Fake ها بیشتر در تست های [feature و Integration](https://sokanacademy.com/blog/10508/software-testing-types) استفاده می شود و معمولا در تست های Unit کاربرد چندانی ندارند. از جمله کاربرد هایی که Fake ها دارند می توان به تست cache، سرویس ایمیل، لاگ و API های خارجی استفاده شده در برنامه اشاره کرد.

سوال3)

درباره تفاوت این دو روش میتوان گفت:

|  |  |
| --- | --- |
| روش Classic | روش Mockist |
| * برای شبیه سازی DOC ها ترجیحا از Stub ها و نمونه های واقعی استفاده می کند. در صورت عدم امکان سراغ ساخت Mock ها و یا Stub های دوگان شده می رود | * حدالامکان از Mock های دوگان شده استفاده می کند و از DOC های واقعی استفاده نمیکند |
| * بیشتر از State Verification استفاده می کند | * بیشتر از Behavior Verification استفاده می کند |
| * امکان استفاده از رویکرد "وسط به بیرون" را فراهم می کند؛ یعنی در توسعه هر قابلیت، از کلاس های تعریف شده در مدل دامنه شروع کرده و توسعه را تا پایان لایه UI ادامه می دهیم. در محیط هایی که وفاداری به مدل دامنه مهم است، این روش مفید است | * امکان استفاده از رویکرد "بیرون به درون" را فراهم میکند؛ یعنی توسعه نرم افزار از بیرونی ترین لایه (مثلا UI) به کمک یک Mock از لایه داخلی تر برنامه شروع شده و به مرور زمان لایه های داخلی تر توسعه داده میشوند |
| * قبل از توسعه هر قابلیت، باید بخشی از تعریف DOC های مربوط به آن انجام شده باشد | * برای توسعه هر قابلیت، نیازی به پیاده شدن هیچ کدام از وابستگی ها نداریم |
| * برای اجرای تست، نیاز به ساخت نمونه از تعداد زیادی از اشیاء وابسته داریم | * برای اجرای تست، صرفا نیاز به ساخت یک Mock حداقلی از اشیاء وابسته داریم |
| * ممکن است خطا های موجود در هر ماژول، روی تست های ماژول های دیگر نیز اثر بگذارد | * نتیجه هر تست صرفا به عملکرد SUT بستگی دارد |
| * به دلیل استفاده از State Verification، تست تا حد خوبی از پیاده سازی مستقل می شود | * به دلیل استفاده از Behavior Verification، تست تا حدی به پیاده سازی وابسته می شود |
| * نیاز به اضافه کردن توابع Getter برای Verify کردن State اشیاء بعد تست داریم | * نیازی به تغییر در Interface کلاس ها برای پیاده سازی تست نداریم |
| * از آنجا که تست ها با کمک اشیاء واقعی انجام می شوند، بهتر میتوان به آنها اعتماد کرد | * برای برنامه های با ساختار لایه ای مناسب تر است. |

به عنوان جمع بندی میتوان گفت هیچ کدام از این دو روش لزوما به دیگری برتری ندارد و انتخاب از بین این دو بیشتر به سلیقه و عادات توسعه ای تیم بستگی دارد.