

به نام خدا

آزمایشگاه مهندسی نرم افزار

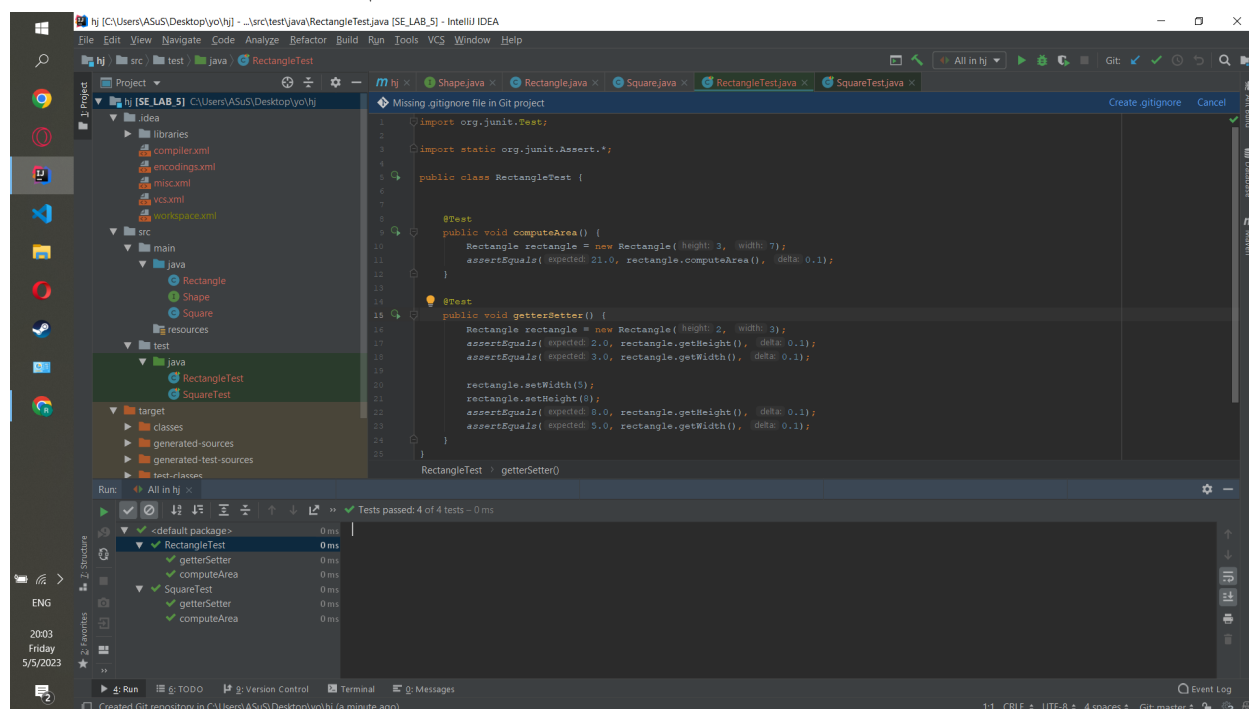
آزمایش پنجم

رضا شکری 97106057
علی کریم لو 97110325

بهار 1402

لینک ریپازیتوری

همانطور که در عکس زیر مشاهده می کنید تمامی 4 تست با موفقیت انجام شده اند.



Interface shape یک ماژول محاسبه مساحت دارد که مربع و مستطیل جداگانه هرکدام برای خود آنرا پیاده سازی میکنند.

در تست ها getter و setter هرکدام از کلاس ها نیز تست شده است.

سوالات

1. اصل اول: Single responsibility principle

این اصل به این معنا نیست که کلاس باید یک کار انجام بدهد، بلکه به این معناست که کلاس باید فقط یک دلیل برای تغییر داشته باشد. در واقع یعنی یک کلاس باید فقط و فقط به یک actor پاسخگو باشد. یعنی حتی اگر کلاس چندین کار هم انجام داد، این باید فقط و فقط در راستای پاسخگویی به یک actor باشد.

اصل دوم: Open/closed principle

یعنی کلاسی که مینویسیم باید برای گسترش دادنش باز باشد، اما برای تغییر دادنش بسته باشد. به این معنی که بتوان به راحتی یک قابلیت جدید به کلاس اضافه کرد، بدون آن که مجبور باشیم کد کلاس را تغییر دهیم. یکی از راه های برآورده کردن این اصل، رعایت اصل DIP است که اصل آخر است.

اصل سوم: Liskov substitution principle

این اصل به این معنا است که کلاس فرزند باید بتواند جایگزین کلاس والد باشد. میدانیم هر کلاسی یک سری pre-condition و یک سری post-condition دارد. این اصل به آن معناست که هر متدی که در کلاس فرزند است و آن را از کلاس والد به ارث برده است، نباید pre-condition قوی تری داشته باشد و همچنین نباید post-condition ضعیف تری داشته باشد.

اصل چهارم: Interface segregation principle

این اصل به این معنی است که یک واسط general خیلی بدتر از چند واسط cohesive است. در واقع بهتر است که یک واسط چندین کار را انجام ندهد.

اصل پنجم: Dependency inversion principle

فرض کنید یک کلاس A و یک کلاس B داریم که بین آنها رابطه association برقرار است، یعنی یک فیلدی از B در A استفاده میشود. این اصل میگوید که بتر است کلاس B کلاسی abstract باشد و concrete نباشد. در واقع کلاس A به specification وابسته باشد و نه به پیاده سازی کلاس B.

2. تعداد زیادی اصول برای مرحله طراحی نرم افزار ها وجود دارد که یکی از آن ها اصول SOLID است. این اصول

به این منظور طراحی شده اند که معماری یک برنامه کاربردی را انعطاف پذیرتر، مقیاس پذیرتر و قابل نگهداری تر کنند. اجرای آنها در کار شما می تواند پیچیدگی کلی پروژه را نیز افزایش دهد.

3. هسته TDD نوشتن تست قبل از هر کد تولید است. این اساساً با نحوه انجام کارها در تضاد است. ما می خواهیم کار را سریع به پایان برسانیم و فقط آنچه را که ارسال می شود بنویسیم. ما کد آزمایشی را ارسال نمی کنیم، بنابراین می خواهیم فقط کد نهایی را بنویسیم و با آن کار تمام شود. تست های نوشتاری، توسعه دهنده را مجبور می کند تا قبل از کدنویسی، یک بار دیگر به تمام الزامات فکر کند، که این چیز خوبی است. اغلب به حذف برخی از فرضیات نامربوط که توسعه دهنده ناخودآگاه مطرح کرده اند کمک می کند. ترجمه الزامات متن به کد اغلب موارد حاشیه ای را نشان می دهد که تیم محصول قبلاً به آنها فکر نکرده بود. با این حال، این کار اضافی از فکر کردن، هنوز هم کار بیشتری بر دوش توسعه دهنده است.

بدون شک TDD سخت است، اما مزایای زیادی دارد. این به ما کمک می کند تا روی نیازمندی ها تمرکز بهتری داشته باشیم، با اطمینان اصلاح کنیم، کد را برای داشتن مسئولیت واحد و رابط های بهتر اجرا کنیم.

4. در صورتی که نیاز به تغییر ابعاد مستطیل ها نداشته باشیم، می توانیم تابع های setter را از کلاس مان حذف کنیم. در این صورت ابعاد مستطیل فقط در هنگام ساخت **object** آن داده می شود و پس از آن قابلیت تغییر ندارد. علاوه بر این در شرایطی که ابعاد مستطیل تغییری نکند، مساحت آن نیز تغییر نمی کند. پس می توانیم در تابع **constructor** درجا مقدار مساحت مستطیل را حساب کرده و به عنوان یک **attribute** ذخیره کنیم تا هر بار فرایند کال شدن تابع و محاسبه مقدار صورت نگیرد.