اصول SOLID

پشت پرده برنامه های ناموفق

برنامه نویسها در حین کدنویسی معمولا سعی می­کنند کد های تمیزی بر اساس دانش و تجربه خودشان بنویسند، اما خیلی وقتها برنامه ها به باگ میخورند. بعد از مدتی هر برنامه ای میخواهد توسعه پیدا کند یا به هر دلیلی نیاز به تغییر دارد، از این رو نمیتوانیم جلوی توسعه را بگیریم

نقص های زیر میتوانند باعث خراب شدن عملکرد یک نرم افزار شوند

1. ایجاد استرس بالا روی یک کلاس با محول کردن وظایف خیلی زیاد
2. ایجاد وابستگی های زیاد بین کلاس ها به طوری تغییر هر کدام از آنها می­تواند روی بقیه تاثیر بگذارد
3. پخش کردن کدهای تکراری در جای جای برنامه

راه حل ها:

1. استفاده از معماری صحیح
2. پیروی از اصول طراحی
3. استفاده از الگوهای طراحی و نوشتن نرم افزار بر اساس آنها

اصول طراحی solid به ما این امکان را میدهد که از یک سری کد tightly coupled و little encapsulation به سمت کدهای loosely coupled و کپسوله شده بر اساس نیاز برویم

اصل تک وظیفه single responsibility principle



Srp میگه " هریک از ماژول های نرم افزار تنها باید یک دلیل برای تغییر داشته باشند".

این بدین معنی است که هر کلاس یا ساختاری تنها باید یک کار برای انجام داشته باشد. هرچیزی در این کلاس باید وابسته به یک "هدف" باشد.

کلاس ما نباید مثل چاقوی تصویر بالا برای هرکاری خودش را تغییر دهد. البته این بدین معنی نیست که کلاس های شما باید شامل یک متد یا یک property باشند.

امکان دارد عضوهای زیادی وجود داشته باشند که فقط یک وظیفه دارند

برای مثال در کد زیر این موضوع رعایت نشده است

1. **public** **class** UserService
2. {
3. **public** **void** Register(**string** email, **string** password)
4. {
5. **if** (!ValidateEmail(email))
6. **throw** **new** ValidationException("Email is not an email");
7. var user = **new** User(email, password);
9. SendEmail(**new** MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });
10. }
11. **public** **virtual** **bool** ValidateEmail(**string** email)
12. {
13. **return** email.Contains("@");
14. }
15. **public** **bool** SendEmail(MailMessage message)
16. {
17. \_smtpClient.Send(message);
18. }
19. }

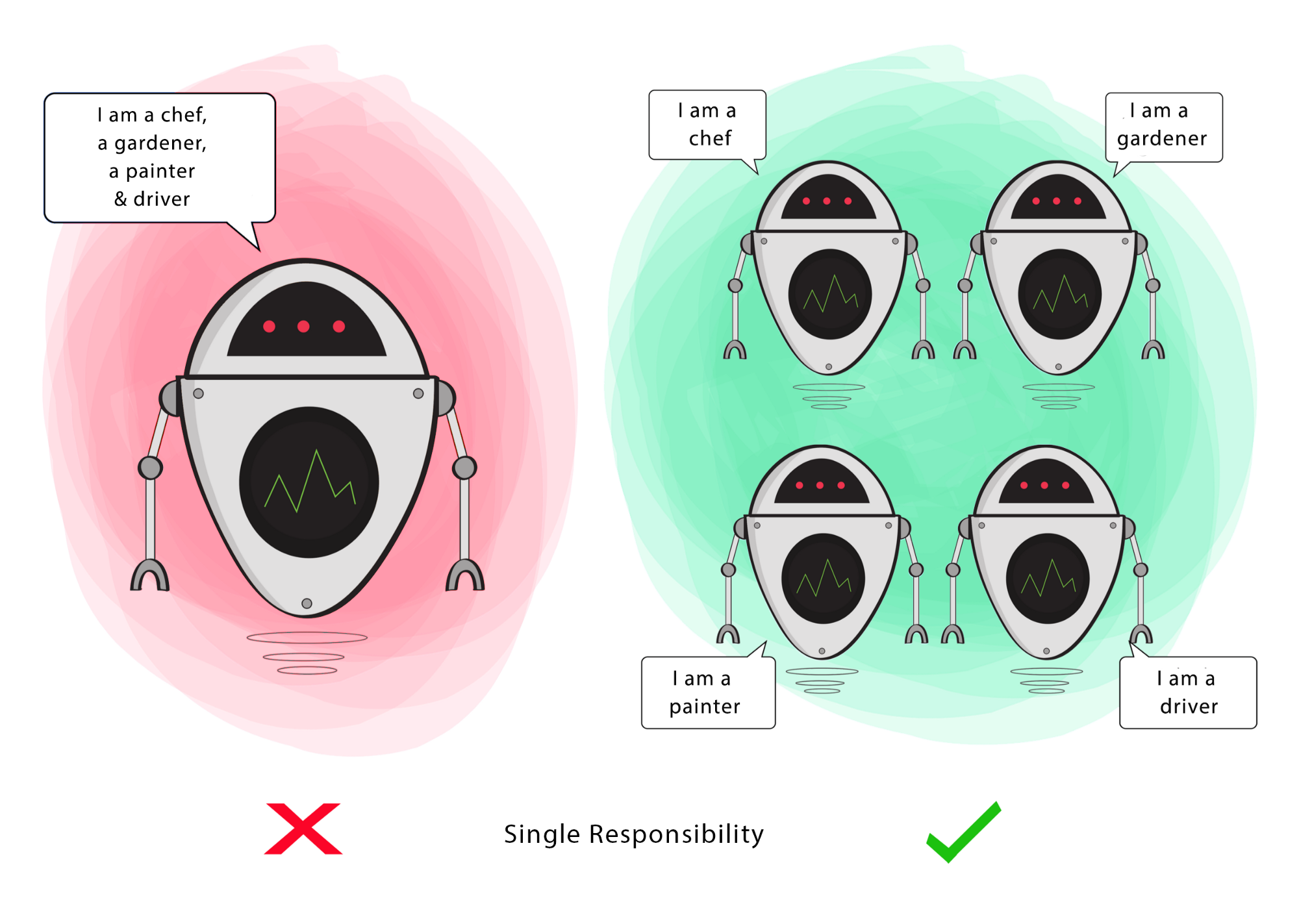
همانطور که دیده می شود، در یک کلا user service هم ثبت نام می کند هم ایمیل میفرستد، برای رعایت srp باید کلاس مربوط ثبت نام از ایمیل زدن جدا شود

1. **public** **class** UserService
2. {
3. EmailService \_emailService;
4. DbContext \_dbContext;
5. **public** UserService(EmailService aEmailService, DbContext aDbContext)
6. {
7. \_emailService = aEmailService;
8. \_dbContext = aDbContext;
9. }
10. **public** **void** Register(**string** email, **string** password)
11. {
12. **if** (!\_emailService.ValidateEmail(email))
13. **throw** **new** ValidationException("Email is not an email");
14. var user = **new** User(email, password);
15. \_dbContext.Save(user);
16. emailService.SendEmail(**new** MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});
18. }
19. }
21. **public** **class** EmailService
22. {
23. SmtpClient \_smtpClient;
24. **public** EmailService(SmtpClient aSmtpClient)
25. {
26. \_smtpClient = aSmtpClient;
27. }
28. **public** **bool** **virtual** ValidateEmail(**string** email)
29. {
30. **return** email.Contains("@");
31. }
32. **public** **bool** SendEmail(MailMessage message)
33. {
34. \_smtpClient.Send(message);
35. }
36. }

همانطور که دیده می شود در مثال بالا یک کلاس مسئول ثبت نام و کلاس دیگری مسئول ارسال ایمیل است.

هدف:

این اصل تلاش می کند که رفتار ها را از هم جدا کند. به این منظور که اگر در اثر تغییر باگی به وجود آمد رفتارهای غیر مرتبط تحت تاثیر قرار نگیرند.



اصل باز و بسته (Open/Closed Principle)

این اصل می گوید که " یک ماژول یا کلاس باید برای گسترش باز و برای تغییر بسته باشد"

"باز بودن برای گسترش" به این معنی است که، ما نیاز به طراحی­ای داریم که تنها در زمانی عملکرد جدید اضافه می شود که نیاز جدیدی به وجود آمده باشد.

"بسته بودن برای تغییر" به این معنی است که اگر یک کلاس را توسعه دادیم و از تست واحد عبور کرد، تغییر نمی کند مگر اینکه به باگ برخورد کند.



در مثال زیر فرض کنید یک کلاس داریم که بیانگر یک مستطیل است و کلاسی داریم که وظیفه محاسبه مساحت مجموعه ای از مستطیل­ها را دارد،