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. }

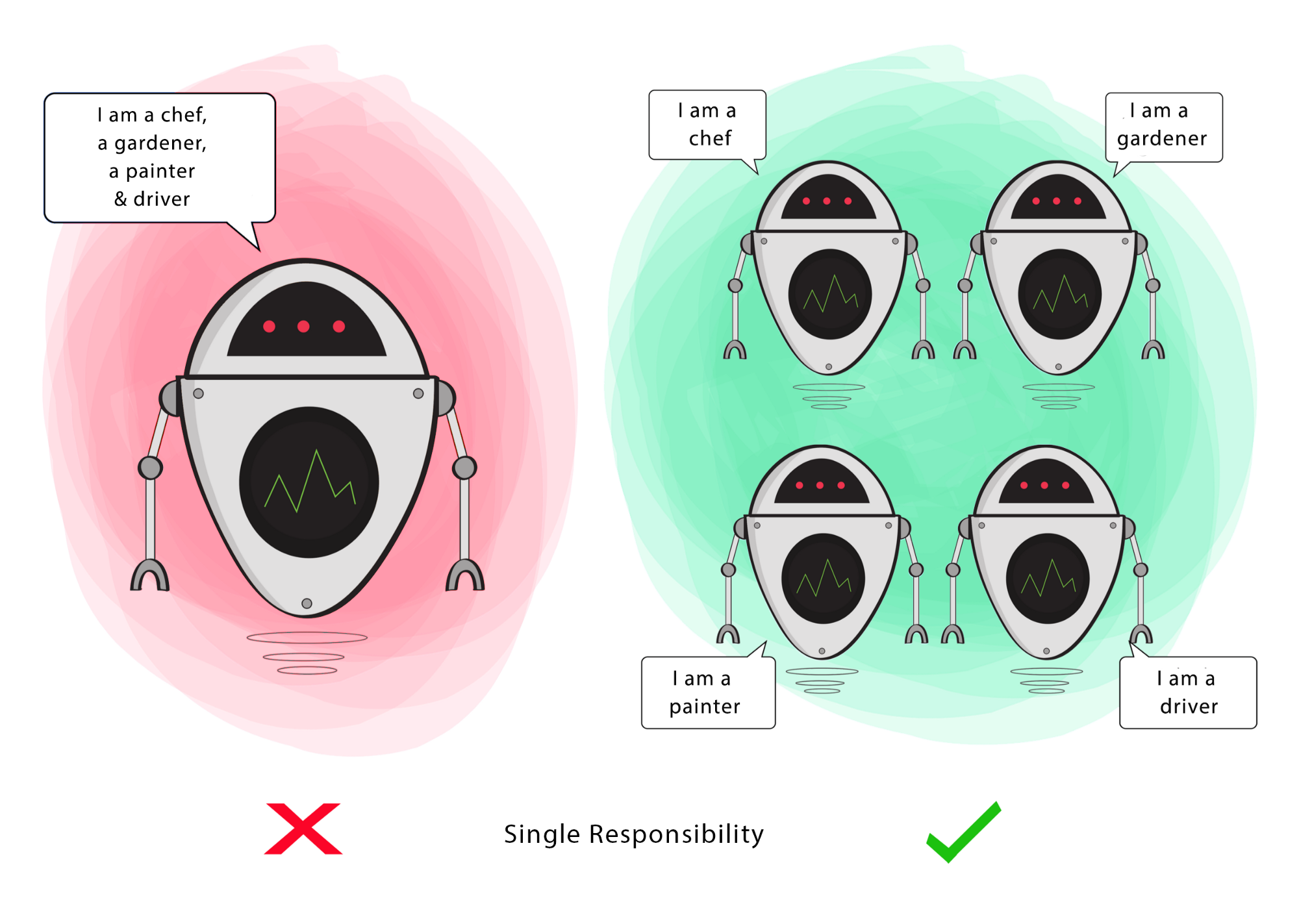
همانطور که دیده می شود، در یک کلا user service هم ثبت نام می کند هم ایمیل میفرستد، برای رعایت srp باید کلاس مربوط ثبت نام از ایمیل زدن جدا شود

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. در این حالت srp هم به طور کامل رعایت شده است.

public class Rectangle

{

public double Height { get; set; }

public double Width { get; set; }

}

public class AreaCalculator

{

public double TotalArea(Rectangle[] arrRectangles)

{

double area=0;

foreach (var objRectangle in arrRectangles)

{

area += objRectangle.Height \* objRectangle.Width;

}

return area;

}

{

در حال حاضر این app هیچ ایرادی ندارد، حال اگر بخواهیم مساحت دایره هم حساب کنیم مجبوریم متد Total Area را تغییر دهیم و در آن چک کنیم که اگر دایره بود چکار کن، اگر مستطیل بود چکار. همین امر باعث زیر سوال بردن بسته بودن نسبت به تغییر می شود.

مجبوریم متدهای خود را به شکل زیر تغییر دهیم:

public double TotalArea(object[] arrObjects)

{

double area = 0;

Rectangle objRectangle = new Rectangle() ;

Circle objCircle;

foreach (var obj in arrObjects)

{

if (obj is Rectangle)

{

area += objRectangle.Height \* objRectangle.Width;

}

else

{

objCircle = (Circle)obj;

area += objCircle.Radius \* objCircle.Radius \* Math.PI;

}

}

return area;

}

اوکی. حال فرض کنیم میخواهیم شی مثلث هم اضافه کنیم، همواره باید این متد را تغییر دهیم.

به طور عمومی راهکار این است که وابسته ها را از سطح کلاس به اینترفیس و کلاسهای abstract تغییر دهیم.

public abstract class Shape

{

public abstract double Area();

{

در این حالت متد محاسبه مساحت به صورت abstract آمده است و هر کدام از کلاس های زیر باید آن را پیاده سازی کنند.

public class Circle : Shape

{

public double Radius { get; set; }

public override double Area()

{

return Radius \* Radius \* Math.PI;

}

}

public class Rectangle : Shape

{

public double Height { get; set; }

public double Width { get; set; }

public override double Area()

{

return Height \* Width;

}

}

public class AreaCalculator

{

public double TotalArea(Shape[] arrShapes)

{

double area = 0;

foreach (var objShape in arrShapes)

{

area += objShape.Area();

}

return area;

}

{

**اصل جایگزینی لیسکوف Liskov Substitution Principle**

اصل جایگزینی لیسکوف می گوید شما باید بتوانید از کلاس های فرزند به جای پدر بدون هیچ تغییری استفاده نمایید. به بیانی دیگر به این نکته تاکید دارد که کلاس فرزند نباید روی رفتارهای پدر تاثیر بگذارد.

این اصل فقط یک اکستنشن از اصل باز و بسته است به این معنی که کلاس فرزند فقط باید کلاس پدر را extend کند، و رفتار آن را تغییر ندهد.

برای مثال اگر پدر دکتر باشد و پسر آن فوتبالیست بشود، پسر نمی تواند جایگزین پدر شود، پس اصل جایگزینی لیسکوف نقض می­گردد.

مثال:

در این مثال، ابتدا میخواهیم ببینیم چگونه اصل جایگزینی لیسکوف زیر سوال می رود، سپس در ادامه آن را اصلاح نماییم. هدف این است که یک app داشته باشیم که داده­ها را توسط گروهی از فایلهای متنی sql مدیریت کنیم. در اینجا نیاز داریم عملکردی را برقرار کنیم که در آن داده­ها قابلیت لود و ذخیره شدن داشته باشند. بنابراین نیاز به کلاسی داریم که این کار را انجام دهد.

1. **public** **class** SqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. **public** **string** SaveText()
10. {
11. /\* Code to save text into sql file \*/
12. }
13. }
14. **public** **class** SqlFileManager
15. {
16. **public** List<SqlFile> lstSqlFiles {**get**;**set**}
18. **public** **string** GetTextFromFiles()
19. {
20. StringBuilder objStrBuilder = **new** StringBuilder();
21. **foreach**(var objFile **in** lstSqlFiles)
22. {
23. objStrBuilder.Append(objFile.LoadText());
24. }
25. **return** objStrBuilder.ToString();
26. }
27. **public** **void** SaveTextIntoFiles()
28. {
29. **foreach**(var objFile **in** lstSqlFiles)
30. {
31. objFile.SaveText();
32. }
33. }
34. }

تا اینجا همه چیز اوکیه. پس از مدتی تصمیمی اتخاذ میگردد مبنی بر اینکه تعدادی فایل فقط خواندنی باشند، پس ما مجبوریم که آنها را محدود کنیم

می توانیم کلاسی تحت عنوان ReadOnlySqlFile بسازیم که از SqlFile ارث بری کند، و متد saveTextIntoFiles را تغییر دهد که از ذخیره شدن در فایل های readonly جلوگیری کند.

1. **public** **class** SqlFile
2. {
3. **public** **string** LoadText()
4. {
5. /\* Code to read text from sql file \*/
6. }
7. **public** **void** SaveText()
8. {
9. /\* Code to save text into sql file \*/
10. }
11. }
12. **public** **class** ReadOnlySqlFile: SqlFile
13. {
14. **public** **string** FilePath {**get**;**set**;}
15. **public** **string** FileText {**get**;**set**;}
16. **public** **string** LoadText()
17. {
18. /\* Code to read text from sql file \*/
19. }
20. **public** **void** SaveText()
21. {
22. /\* Throw an exception when app flow tries to do save. \*/
23. **throw** **new** IOException("Can't Save");
24. }
25. }

بنابراین باید کلاس SqlFileManager را با اضافه کردن چند شرط تغییر دهیم.

1. **public** **class** SqlFileManager
2. {
3. **public** List<SqlFile? lstSqlFiles {**get**;**set**}
4. **public** **string** GetTextFromFiles()
5. {
6. StringBuilder objStrBuilder = **new** StringBuilder();
7. **foreach**(var objFile **in** lstSqlFiles)
8. {
9. objStrBuilder.Append(objFile.LoadText());
10. }
11. **return** objStrBuilder.ToString();
12. }
13. **public** **void** SaveTextIntoFiles()
14. {
15. **foreach**(var objFile **in** lstSqlFiles)
16. {
17. //Check whether the current file object is read-only or not.If yes, skip calling it's
18. // SaveText() method to skip the exception.
20. **if**(! objFile **is** ReadOnlySqlFile)
21. objFile.SaveText();
22. }
23. }
24. }

در اینجا ما متد SaveTextIntoFiles را در SqlFileManager تغییر دادیم که مشخص میکند، آیا نمونه ای ReadOnlySqlFile هست یا نه .

در اینجا ما نمیتوانیم از ReadOnlySqlFile بدون تغییر SqlFileManager به جای پدرش استفاده کنیم. پس می­توانیم بگوییم این طراحی از LSP پیروی نمی­کند. برای رفع این مشکل به روشی که در ادامه توضیح داده می­شود عمل می کنیم.

ابتدا دو اینترفیس ایجاد می­کنیم.

1. **public** **interface** IReadableSqlFile
2. {
3. **string** LoadText();
4. }
5. **public** **interface** IWritableSqlFile
6. {
7. **void** SaveText();
8. }

حال این اینترفیس را پیاده سازی می­کنیم.

1. **public** **class** ReadOnlySqlFile: IReadableSqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. }
10. **public** **class** SqlFile: IWritableSqlFile,IReadableSqlFile
11. {
12. **public** **string** FilePath {**get**;**set**;}
13. **public** **string** FileText {**get**;**set**;}
14. **public** **string** LoadText()
15. {
16. /\* Code to read text from sql file \*/
17. }
18. **public** **void** SaveText()
19. {
20. /\* Code to save text into sql file \*/
21. }
22. }

حال طراحی کلاس SqlFileManager به صورت زیر می­شود

1. **public** **class** SqlFileManager
2. {
3. **public** **string** GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
4. {
5. StringBuilder objStrBuilder = **new** StringBuilder();
6. **foreach**(var objFile **in** aLstReadableFiles)
7. {
8. objStrBuilder.Append(objFile.LoadText());
9. }
10. **return** objStrBuilder.ToString();
11. }
12. **public** **void** SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
13. {
14. **foreach**(var objFile **in** aLstWritableFiles)
15. {
16. objFile.SaveText();
17. }

در اینجا متد GetTextFromFiles() فقط لیستی از نمونه هایی را می گیرد که اینترفیس IReadOnlySqlFile را پیاده سازی کرده باشند. و همچین متد SaveTextInrtoFiles فقط لیستی از می­گیرد که اینترفیس IWritableSqlFile را پیاده سازی کرده باشند. به بیانی دیگر، در این مثال SqlFile ها.

## اصل جداسازی اینترفیس ها Interface Segregation Principle (ISP)

این اصل بیان میکند که کلاینت­ها نباید مجبور به پیاده سازی اینترفیسهایی باشند که آنها نیازی ندارند. به جای یک اینترفیس چاق، چندین اینترفیس کوچک داشته باشیم که گروهی از متدها درون خود داشته باشند.

این میتواند به این معنی باشد که اینترفیس­ها باید در وهله­ی اول از SRP پیروی کنند.

فرض کنید میخواهیم یک سیستم داشته باشیم که در آن برای یک شرکت IT که شامل نقشهای TeamLead و Programmer است که در آن TeamLead تسک های بزرگ را به تسک­های کوچک تبدیل می کند، و آن ها را به برنامه نویس­های خودش assign میکند.

1. **public** Interface ILead
2. {
3. **void** CreateSubTask();
4. **void** AssginTask();
5. **void** WorkOnTask();
6. }
7. **public** **class** TeamLead : ILead
8. {
9. **public** **void** AssignTask()
10. {
11. //Code to assign a task.
12. }
13. **public** **void** CreateSubTask()
14. {
15. //Code to create a sub task
16. }
17. **public** **void** WorkOnTask()
18. {
19. //Code to implement perform assigned task.
20. }
21. }

تا اینجا همه چیز خوب است. حال فرض کنید یک نقش به عنوان Manager وارد تیم می شود، ولی فقط می خواهد به TeamLead تسک assign کند، و قرار نیست روی تسکی کار کند. کلاس Manager اینترفیس ILead را به صورت زیر پیاده سازی می کند، به صورتی که نتواند روی تسک کار کند.

1. **public** **class** Manager: ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a task.
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. **throw** **new** Exception("Manager can't work on Task");
14. }
15. }

همانطور که دیده می­شود کلاس Manager مجبور به پیاده سازی متدی شد که اصلا نیازی به آن ندارد. و این اصل ISP را زیر سوال می برد . برای رفع این موضوع ، یک اینترفیس IProgrammer می­سازیم که تنها امضای متد WorkOnTask در آن آمده است، و یک اینترفیس ILead می سازیم که در آن AssignTask, CreateSubTask آمده است.

1. **public** **interface** IProgrammer
2. {
3. **void** WorkOnTask();
4. }
5. **public** **interface** ILead
6. {
7. **void** AssignTask();
8. **void** CreateSubTask();
9. }
10. **public** **class** Programmer: IProgrammer
11. {
12. **public** **void** WorkOnTask()
13. {
14. //code to implement to work on the Task.
15. }
16. }
17. **public** **class** Manager: ILead
18. {
19. **public** **void** AssignTask()
20. {
21. //Code to assign a Task
22. }
23. **public** **void** CreateSubTask()
24. {
25. //Code to create a sub taks from a task.
26. }
27. }

همانطور که دیده می شود دیگر manager مجبور به پیاده سازی متد اضافی نیست.حال کلاس TeamLead باید از هردو اینترفیس ILead,IProgrammer پیروی کند.

1. **public** **class** TeamLead: IProgrammer, ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a Task
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task from a task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. //code to implement to work on the Task.
14. }
15. }

در اینجا دیده می شود که هدف و مسئولیت­ها به خوبی از هم جدا شده اند.