SOLID principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

SOLID design principles in C# are basic design principles.

S Single Responsibility Principle (SRP)

O Open closed Principle (OSP)

L Liskov substitution Principle (LSP)

I Interface Segregation Principle (ISP)

D Dependency Inversion Principle (DIP)

Developers start building applications with good and tidy designs using their knowledge and experience. But over time, applications might develop bugs. The application design must be altered for every change request or new feature request. After some time we might need to put in a lot of effort, even for simple tasks and it might require full working knowledge of the entire system. But we can't blame change or new feature requests. So who is the culprit here? Obviously it is the design of the application.

1. Putting more stress on classes by assigning more responsibilities to them. (A lot of functionality not related to a class.)
2. Forcing the classes to depend on each other. (in other words tightly coupled)
3. Spreading duplicate code in the system/application.

S: Single Responsibility Principle

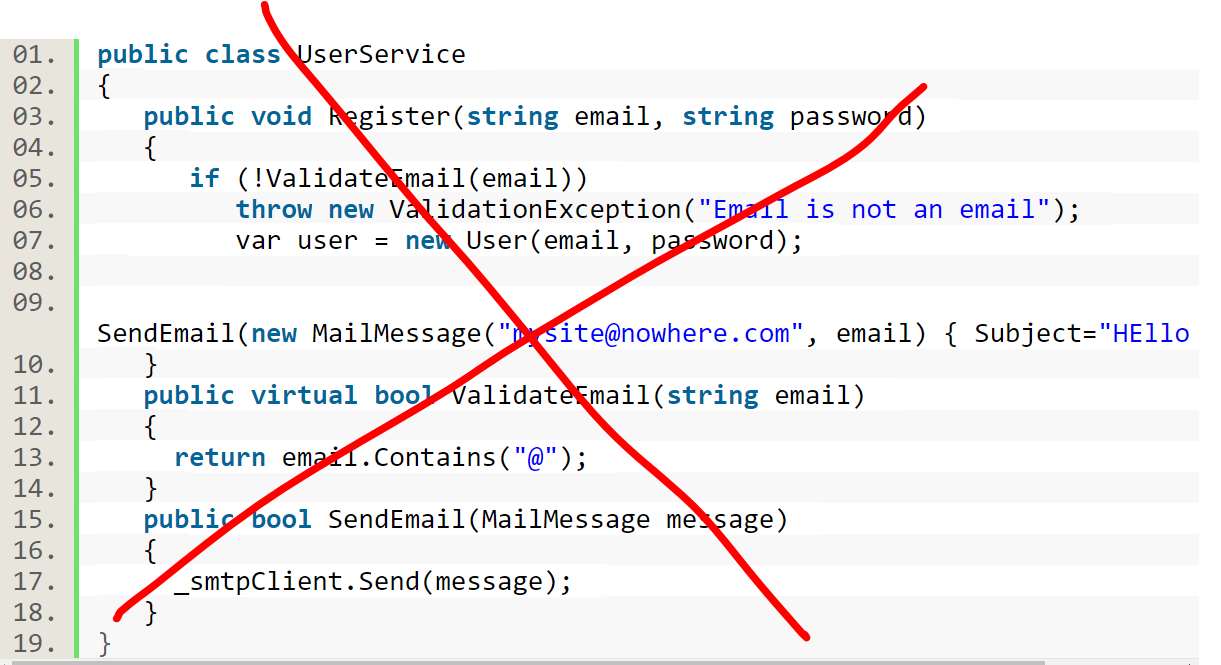
This means that every class, or similar structure, in your code should have only one job to do.

Our class should not be like a Swiss knife.

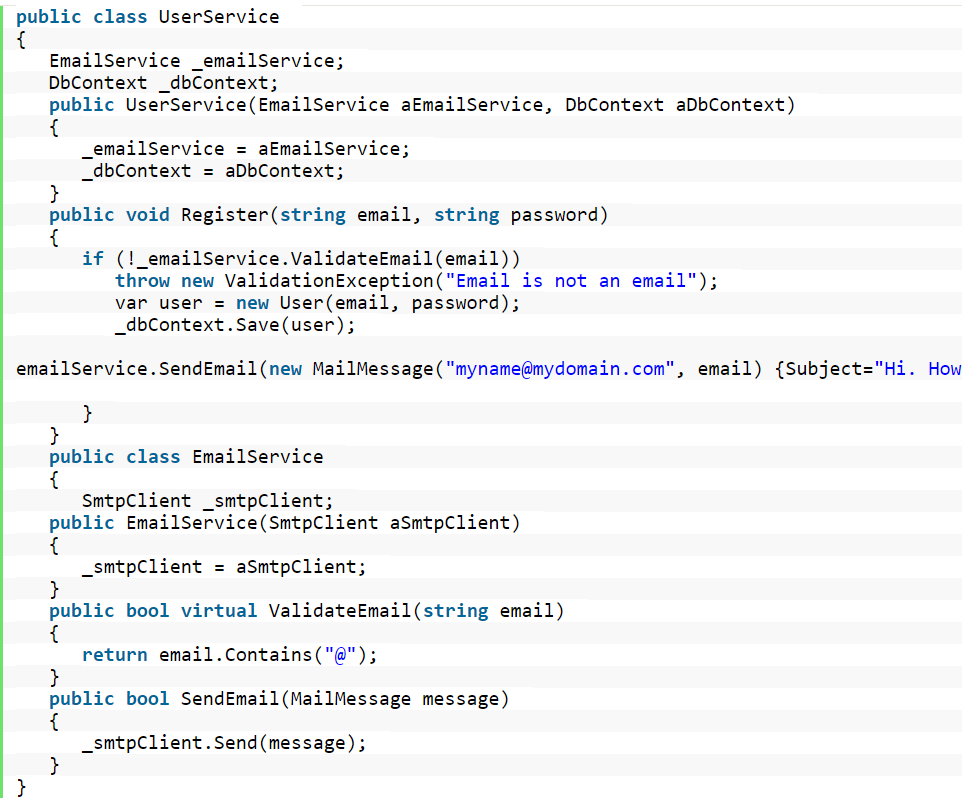
There may be many members as long as they relate to single responsibility.

A good separation of responsibilities is done only when we have the full picture of how the application should work.

The following looks fine, but it is not following SRP.



Instead­



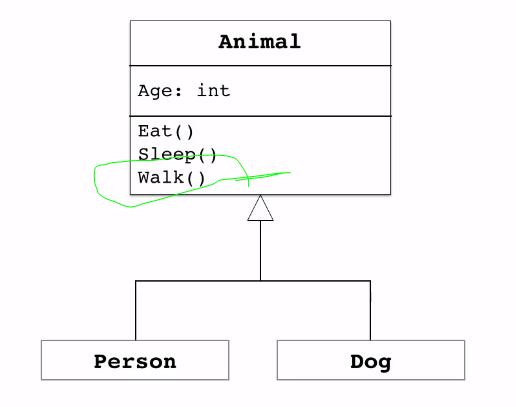
O: Open/Closed Principle

The Open/closed Principle says "A software module/class is open for extension and closed for modification"

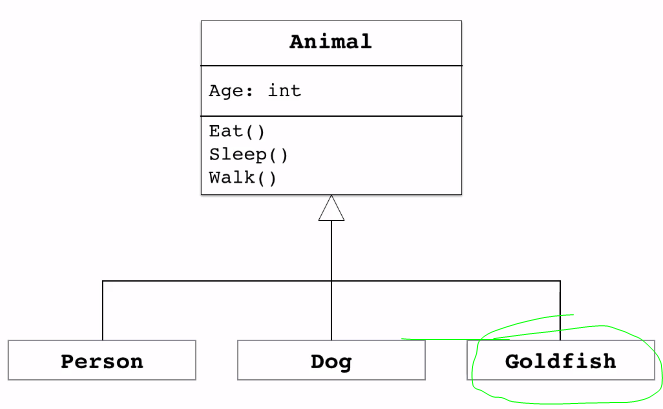
"Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated.

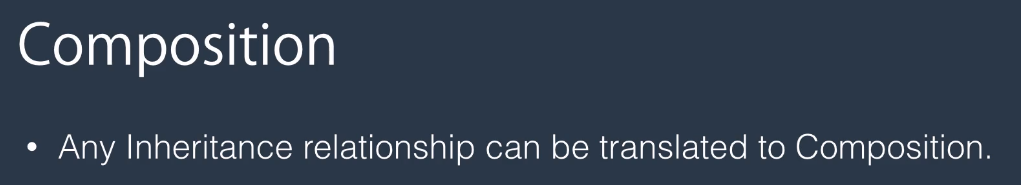
"Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this. But better is use composition.

Per es se aggiungiamo il metodo walk le classi dipendenti potrebbero dover essere cambiate o ricompialte e rideploiate:

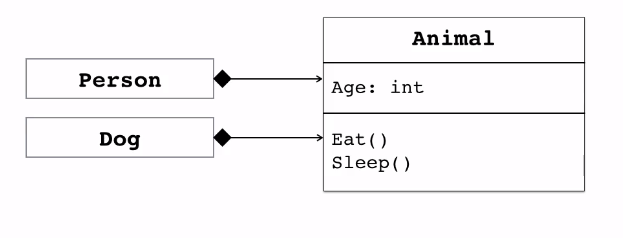


Oppure andiamo ad inserire il pesce, il problema è che il pesce non cammina percui dobbiamo andare a ridefinire l’ereditarietà:

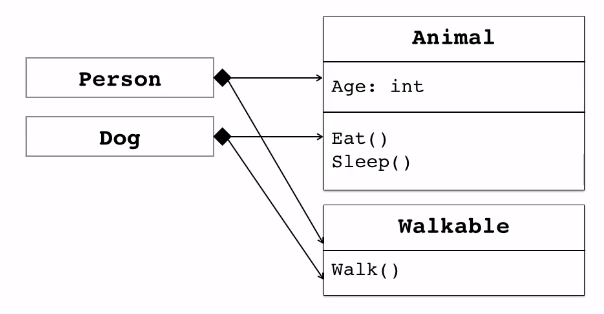


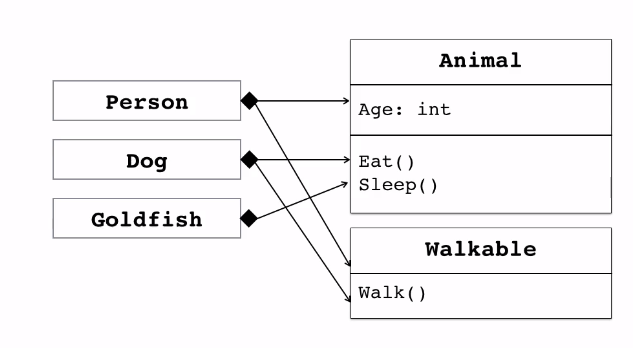


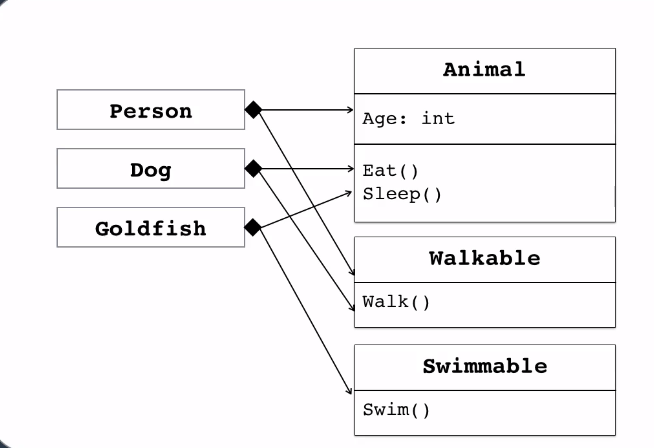
Qui person e dog hanno un animale:



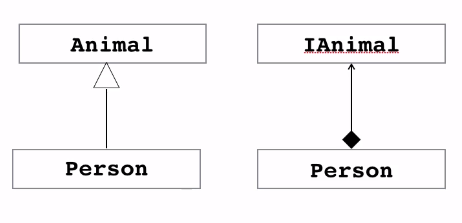
Supponiamo di avere dimenticato di inserire l’abilità di camminare, come possiamo rimediare senza creare impatti?:

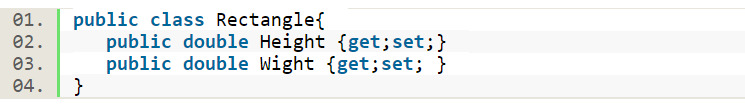


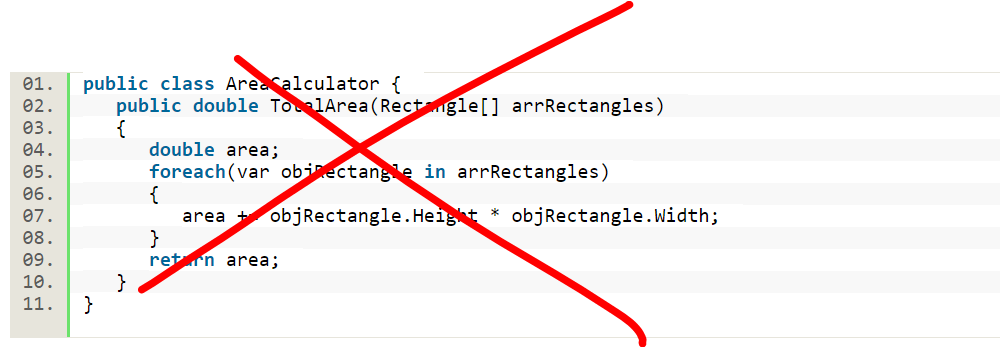




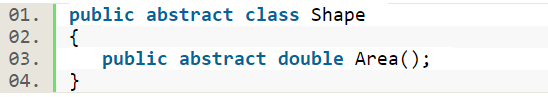
In questo caso usare l’ereditarietà o la composizione sembrerebbe uguale ma un altro beneficio è che possiamo usare l’interfaccia con la composizione…:

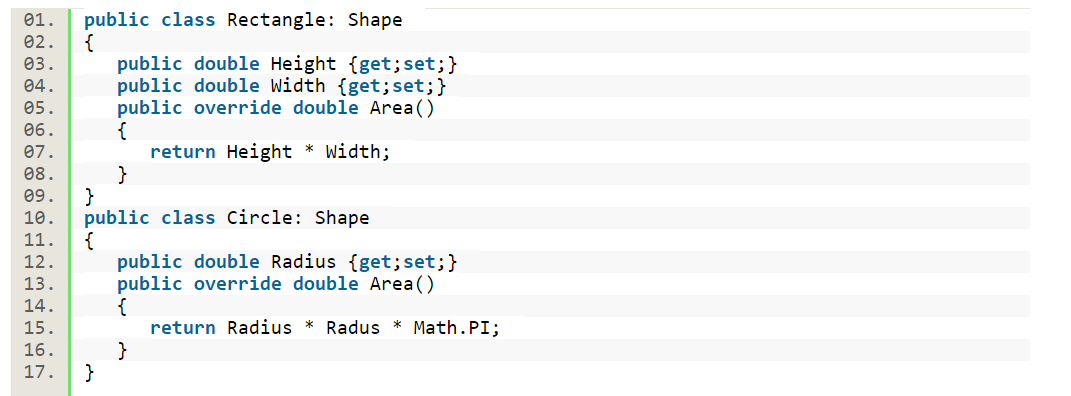


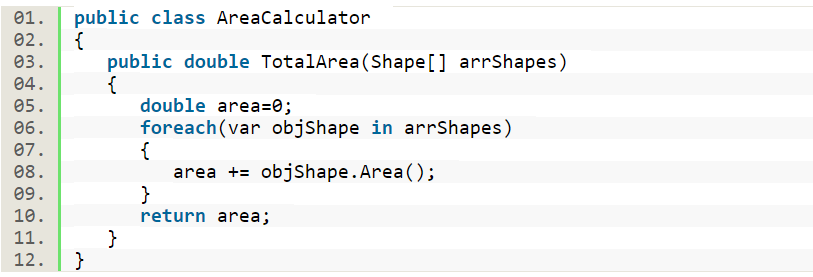




But can we extend our app so that it could calculate the area of not only Rectangles but also the area of Circles as well?



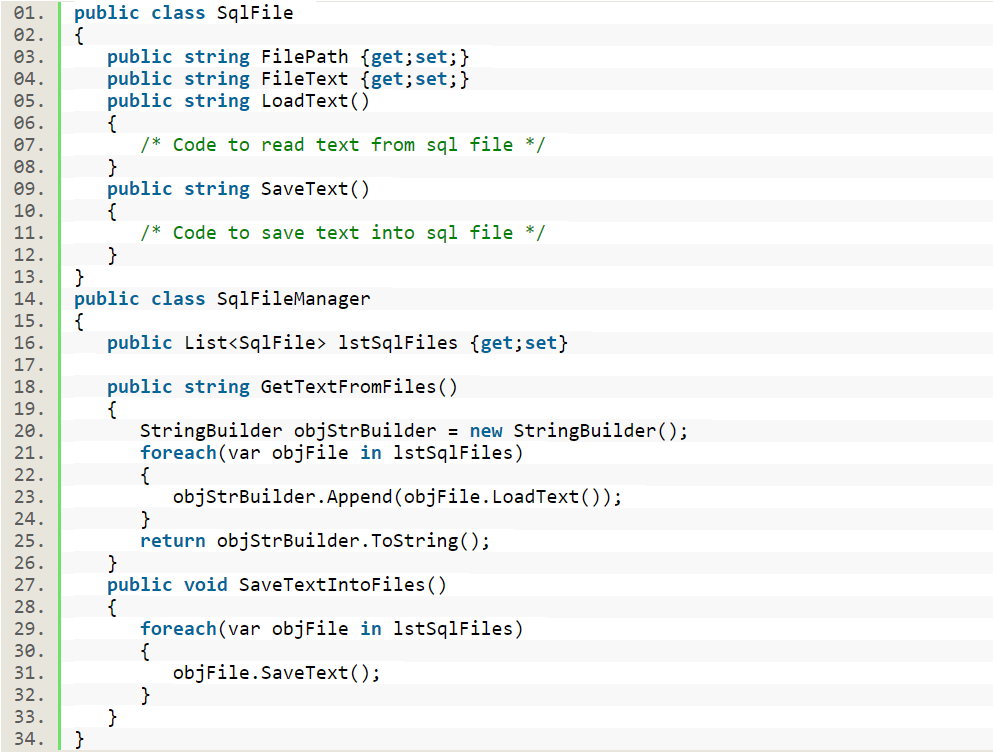




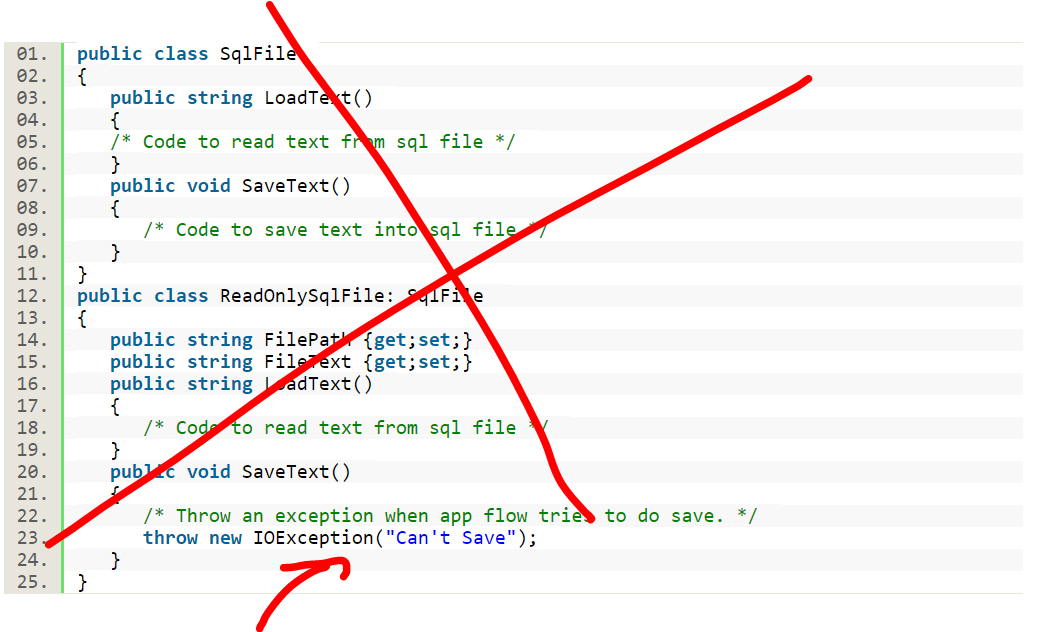
L: Liskov Substitution Principle

we must ensure that new derived classes extend the base classes without changing their behavior. A father is a doctor whereas his son wants to become a cricketer. So here the son can't replace his father even though they both belong to the same family hierarchy.

Suppose we need to build an app to manage data using a group of SQL files text. Here we need to write functionality to load and save the text of a group of SQL files in the application directory. So we need a class that manages the load and saves the text of group of SQL files along with the SqlFile Class.

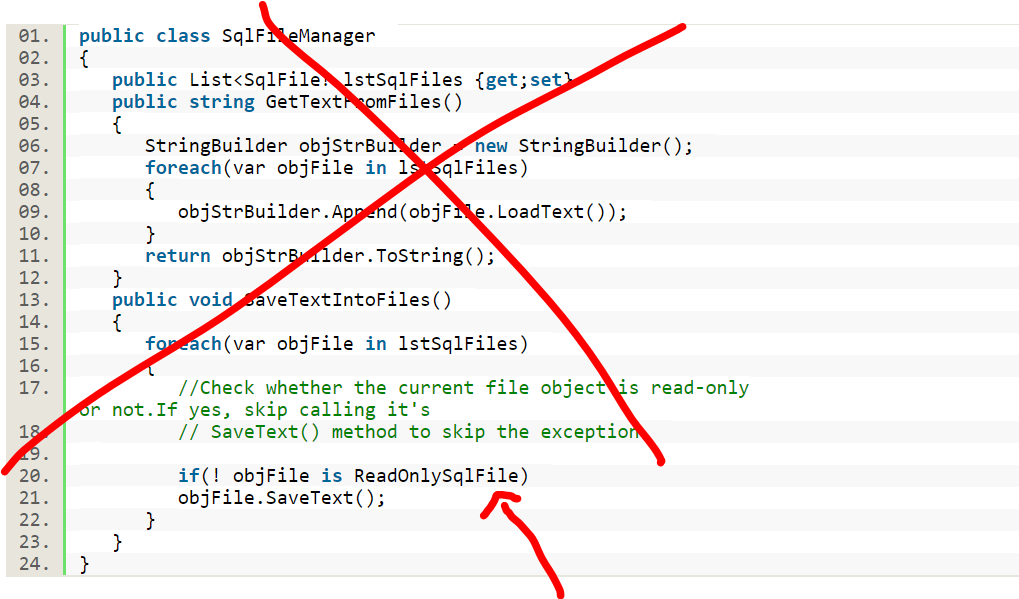


After some time our leaders might tell us that we may have a few read-only files in the application folder, so we need to restrict the flow whenever it tries to do a save on them.

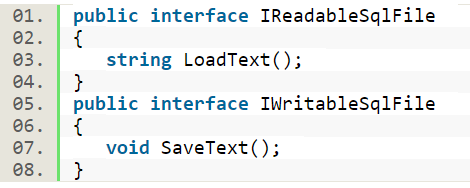


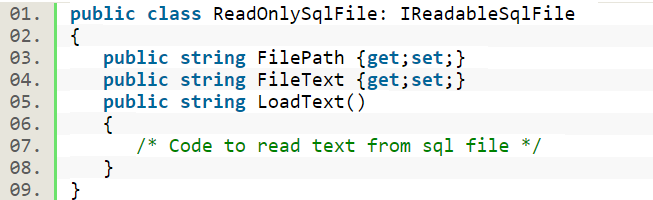
Or to avoid to throw exception:

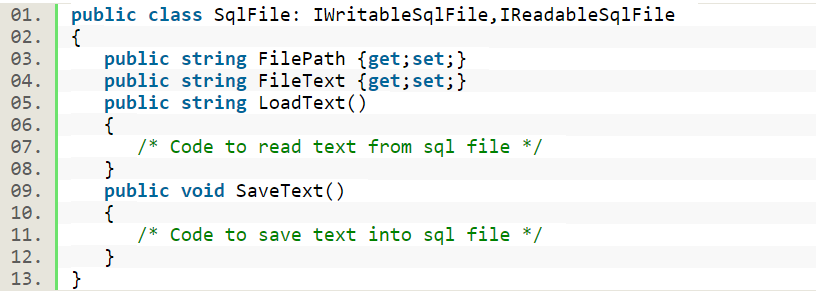
To avoid an exception we need to modify "SqlFileManager" by adding one condition to the loop.

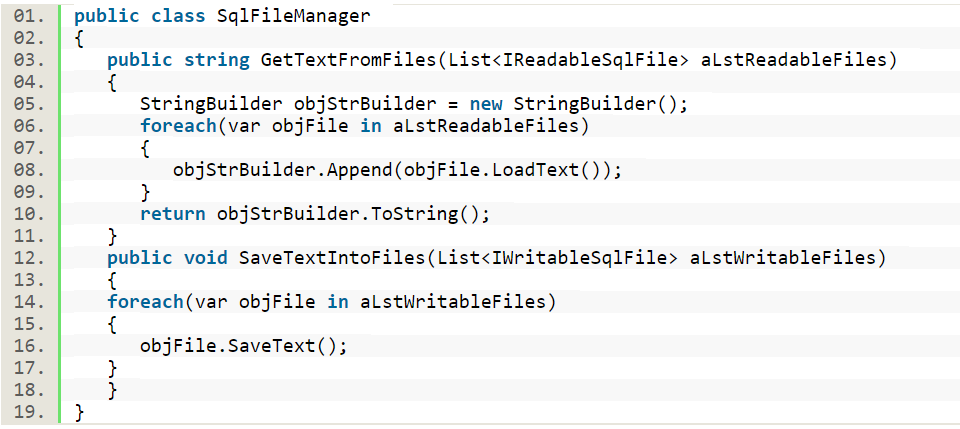


This is the solution:





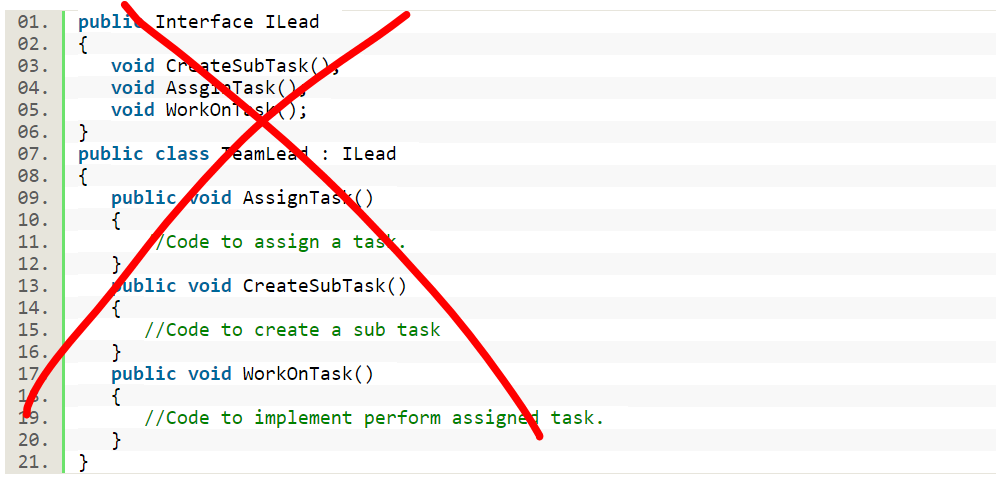


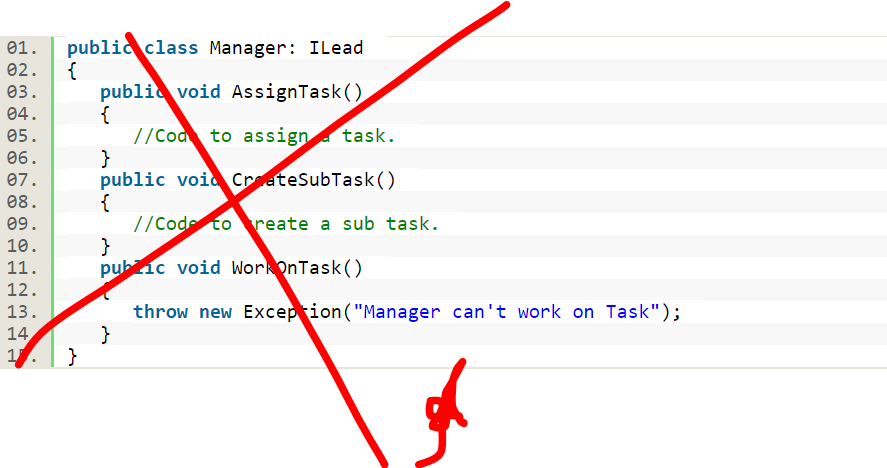


I Interface Segregation Principle.

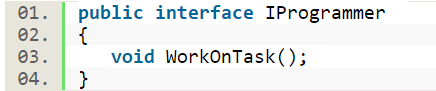
The Interface Segregation Principle states "that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.".

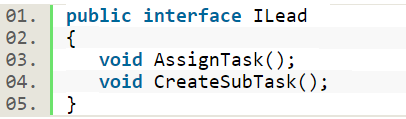
Suppose we need to build a system for an IT firm that contains roles like TeamLead and Programmer where TeamLead divides a huge task into smaller tasks and assigns them to his/her programmers or can directly work on them.

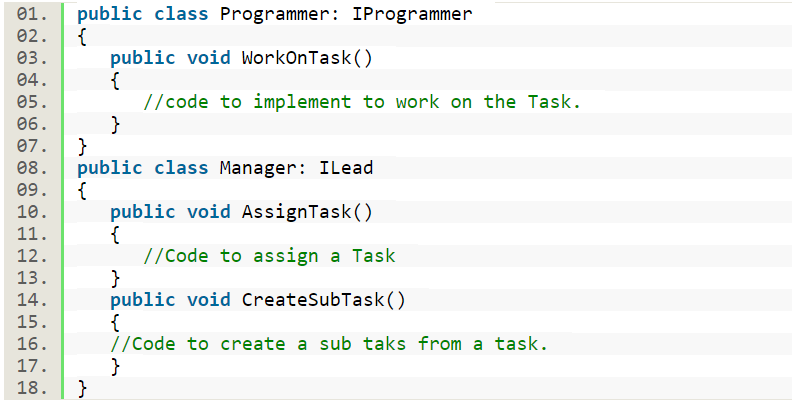


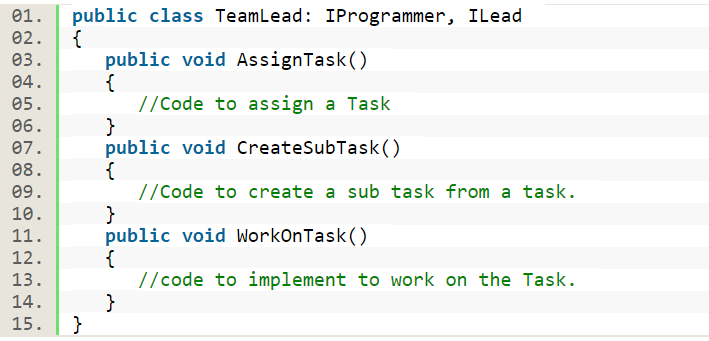


The solution:







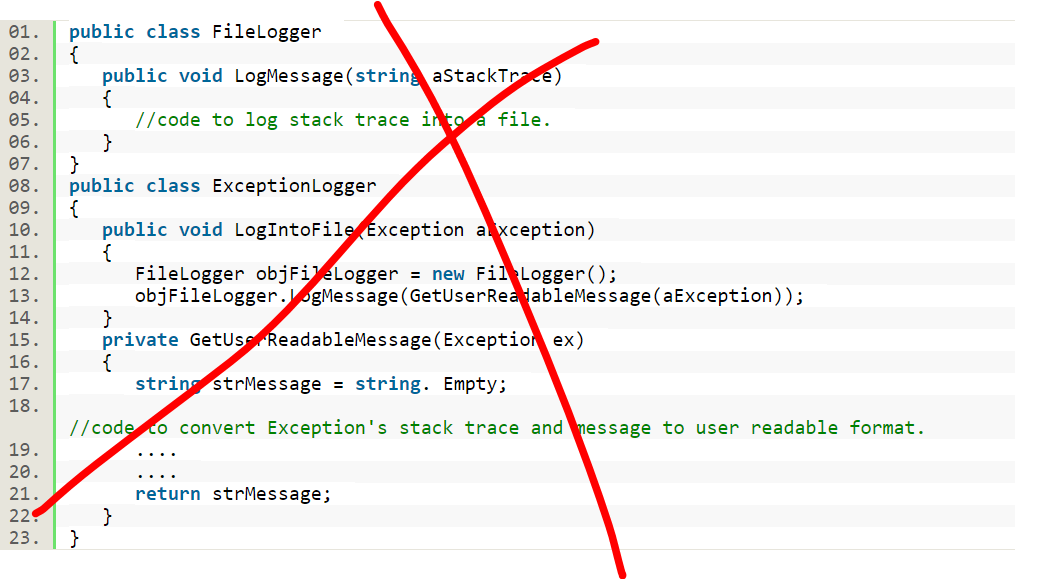


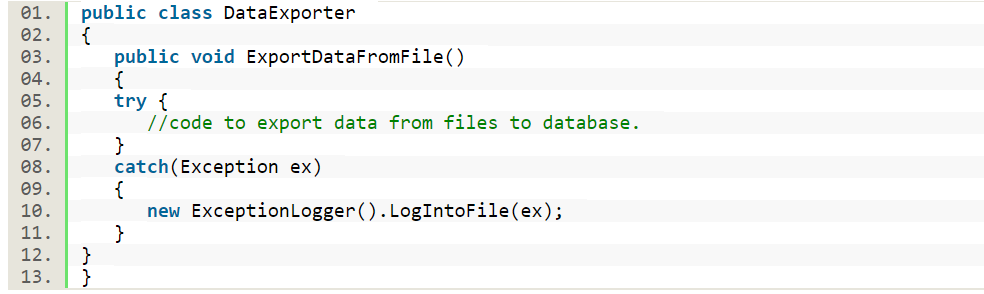
Ma come facciamo ad iniettare questa classe??

D: Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

Suppose we need to work on an error logging module that logs exception stack traces into a file. Simple, isn't it? The following are the classes that provide the functionality to log a stack trace into a file.





But our client wants to store this stack trace in a database if an IO exception occurs.

