S.O.L.I.D Nedir?

S.O.L.I.D, yazılım geliştiriken sürdürülebilir kod yazmamızı sağlayan bir prensipler bütünüdür.

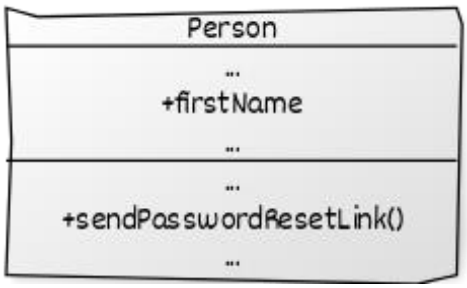
- Buradaki sürdürülebilirlikten kasıt; yazılım gereksinimleri değiştiğinde ya da mevcut yazılıma eklemeler yapıldığında sistemin buna direnç göstermemesi, en azından en az direnci göstermesi yani esnek olmasıdır. Bunların yanı sıra bakımının ve anlaşılmasının kolay olması gibi nedenler de sayılabilir.

Bunları yapmamızı sağlayan 5 maddeyi inceleyeceğiz.

1.Single Responsibility Princible (Tek Sorumluluk Prensibi)

Her sınıf, metot, fonksiyon tek bir sorumluluğa sahip olmalıdır.

- Şayet bu kurala uymazsak ilerleyen süreçte bir değişikliğe gidildiğinde bunun etkisini birçok yerde görmüş oluruz. Nedeni ise bir yapıya birden fazla sorumluluk yüklenmesinden dolayıdır. Eğer değişikliklerden etkilenen yerler arasında sistemin birçok yerinde kullanılan bir yapımız da varsa maliyet gittikçe artacaktır.

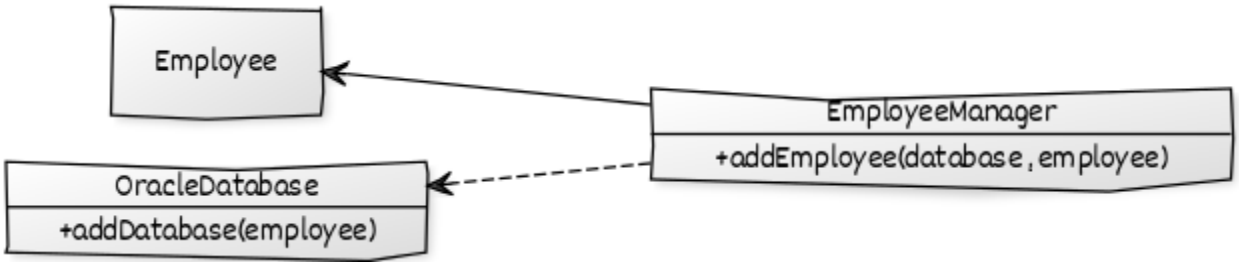


- Yukarıdaki diyagrama ve koda baktığımızda `Person` sınıfı içerisinde `sendPasswordResetLink()` diye bir metot bulunmaktadır. Bu sınıfın asıl amacı kişilere ait bilgileri tutmaktır, şifre sıfırlama bağlantısı göndermek değil. Birden fazla sorumluluk yüklendiği için olası bir mail gönderme değişikliğinde bu sınıf da etkilenecektir.
- Yukarıdaki UML diyagramını biraz daha düzenlersek aşağıdaki gibi bir yapı elde edilir.

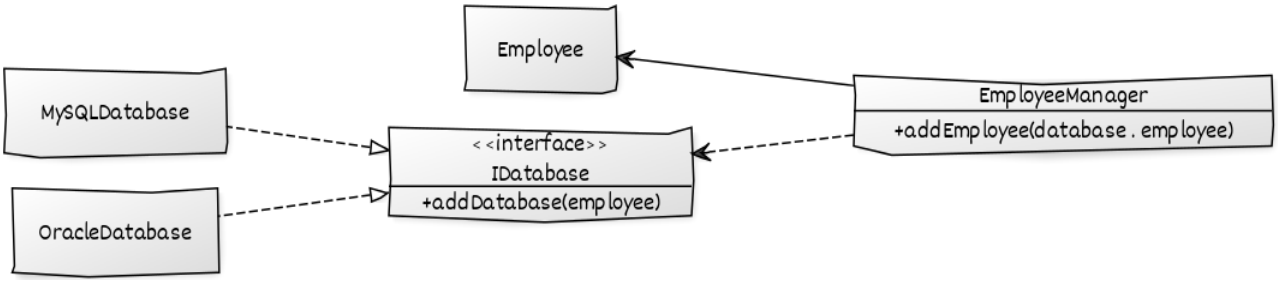
2.Open Close Princible (Açık Kapalı Prensibi)

Yapılarımız (sınıf, metot, fonksiyon) gelişime açık değişime kapalı olmalıdır.

- Yazılımlar için zamanla değişim şüphesiz kaçınılmazdır; değişen iş kuralları, kullanılan harici kütüphaneler gibi başlıca nedenler örnek gösterilebilir. Bu prensibin anlatmak istediği şey **yeni bir davranış ya da özellik eklemek istediğimiz durumda; yapmak istediğimiz değişikliği mevcut koda dokunmadan, değişimi sadece yeni kodlar üzerinden sağlamaktır.**



- Yukarıdaki UML diyagramını biraz daha düzenlersek aşağıdaki gibi bir yapı elde edilir. Yeni bir eklemede mevcut koda dokunmaya gerek kalmıyor bu sayede. Kayıt işlemlerini `MySQL` üzerinde yapmak istediğimiz zaman `MySQLDatabase` adında bir sınıf oluşturup `IDatabase` arayüzünü uygulamamız yeterlidir.

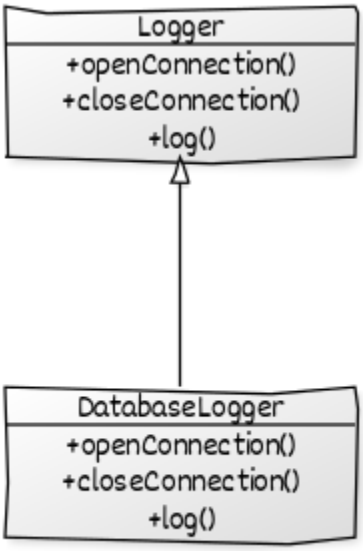


3. Liskov Substitution Princible (Liskov un Yerine Geçme Prensibi)

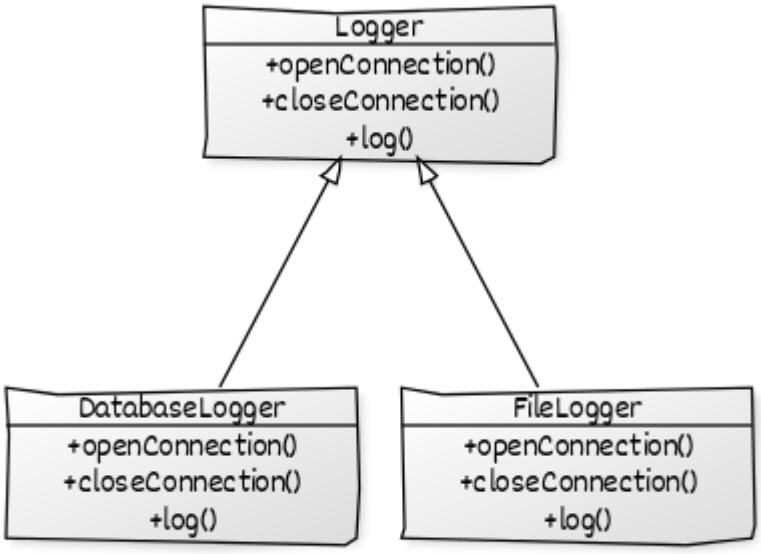
Alt sınıflardan oluşan nesnelerin, üst sınıfın nesneleri ile yer değiştirdiklerinde aynı davranışı sergilemesi gerekmektedir.

- Alt sınıflar, üst sınıflardan türediği için onların davranışlarını devralırlar. Eğer üst sınflara ait davranışları gerçekleştirmiyorlarsa davranışı yapan metotu muhtemelen boş bırakır ya da bir hata fırlatırız fakat bu işlemler kod kirliliğine ve

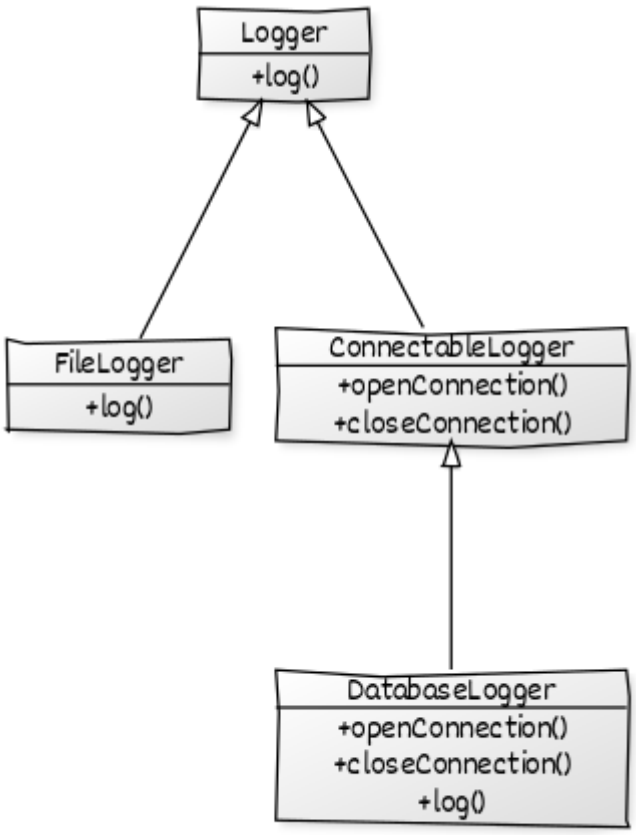
gereksiz kod kalabalığına neden olmaktadır. Bunların yanı sıra projeye daha sonradan dahil olacak geliştiriciler için de sorun oluşturmaktadır. Geliştirici, sistemin sağlıklı yürüdüğünü düşünerek gerçekleştirilmeyen bir davranışı kullanmaya çalışabilir.



- Başlangıç aşaması için bir problem görünmezken ilerleyen zamanlarda veri tabanı değil de bir dosyaya kayıt işlemi alınacağı zaman aşağıdaki gibi bir görünüm meydana gelecektir.



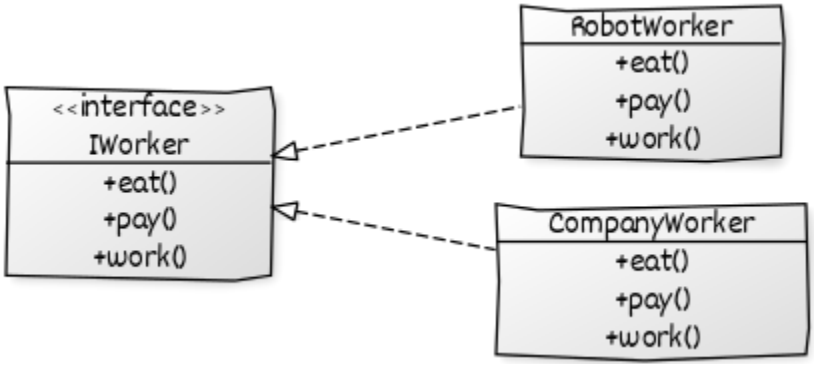
- Bağlantı açma ve kapatma işlemleri veri tabanına aittir, bir dosyaya değil. Gereksiz hata fırlatmaları, kodun okunmasındaki zorluk, kod kalabalığı gibi birçok olaya neden olmaktadır. Burada bu işlemler bir ara sınıfa alınabilir.



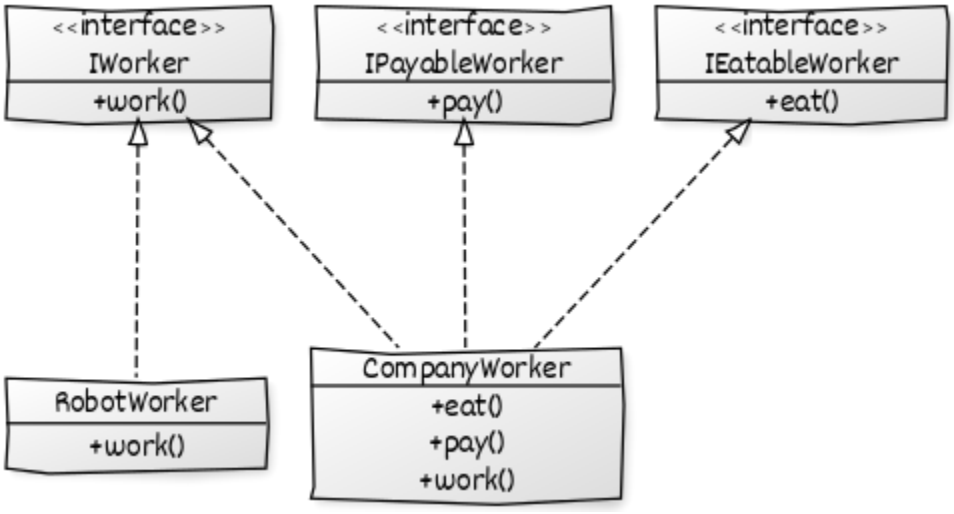
4.Interface Segregation Princible (Arayüz Ayrımı Prensibi)

Sınıflar, kullanılmadığı metotları içeren arayüzleri uygulamaya zorlanmamalıdır.

- Arayüzlerimizde genel olarak birçok operasyonel işlem barındırabiliriz fakat bu arayüzü uygulayan sınıfların, bazılarını kullanmama durumu olabilmektedir. **Bir sınıf birden fazla arayüzü uygulaması özelliğiyle de birlikte bu prensip, bu tür durumlarda arayüzlerin ayrılmasını ve ihtiyaç halinde olanların kullanmasını söylemektedir**.



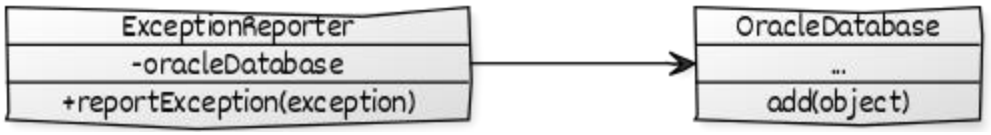
- Yukarıdaki diyagram incelendiğinde, şirket çalışanları `IWorker` arayüzünü uygulamaktadır; yemek yeme, ödeme alma, çalışma gibi davranışları gerçekleştirmektedir. Fakat daha sonradan bazı işler robotlar tarafından yapılmaya başlandı ya da dış kaynaktan birileri(outsource) de çalışmaya başladı. Bu durumda bazı davranışlar gerçekleşmeyecektir. Örneğin robotların yemek yeme ya da ödeme alma davranışını gerçekleştirememesi gibi ya da dış kaynaktan gelenlere verilmeyen yemek imkanı. Bu gerçekleşmeyen davranışların içlerini ya boş bırakma ya da hata fırlatma durumunda kalırız. **Bu tür durumlarda bu prensip bizlere bu arayüzlerin ayrılmasını ve ihtiyaç halinde olanların kullanılmasını söylemektedir.**
- Yukarıdaki UML diyagramını biraz daha düzenlersek aşağıdaki gibi bir yapı elde edilir. `Work()`, `pay()`, `eat()` davranışları başka arayüzlere aktarıldı ve ihtiyaç halinde olanlar uygulandı.



5.Dependeny Inversion Princible (Bağımlılıkların Tersine Çevrilmesi Prensibi)

Yüksek seviye sınıflar, düşük seviye sınıflara bağlı olmamalıdır. Her ikisi de soyutlamalara bağlı olmalıdır.

Soyutlamalar, detaylara bağlı olmamalıdır. Detaylar, soyutlamalara bağlı olmalıdır.



- Yukarıdaki diyagram incelendiğinde `ExceptionReporter` sınıfının (yüksek seviyeli sınıf), `OracleDatabase` sınıfına (düşük seviyeli sınıf) direkt olarak bağımlı olduğu görülmektedir. İleride veri tabanı olarak Oracle değil de MySQL kullanmak istersek maalesef bu sınıfa müdahale etmek zorunda kalacağız. Bu istenmeyen bir davranıştır. Bunun çözümünü ise buradaki **bağımlılıkları soyutlayarak** sağlayacağız.
- Yukarıdaki UML diyagramını biraz daha düzenlersek aşağıdaki gibi bir yapı elde edilir.