Dependency Inversion

By Steve Faurie

Dependency Inversion

Described in Agile Principles, Patterns and Practices in C# by Robert C. Martin

Dependency Inversion Basic Principles

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions

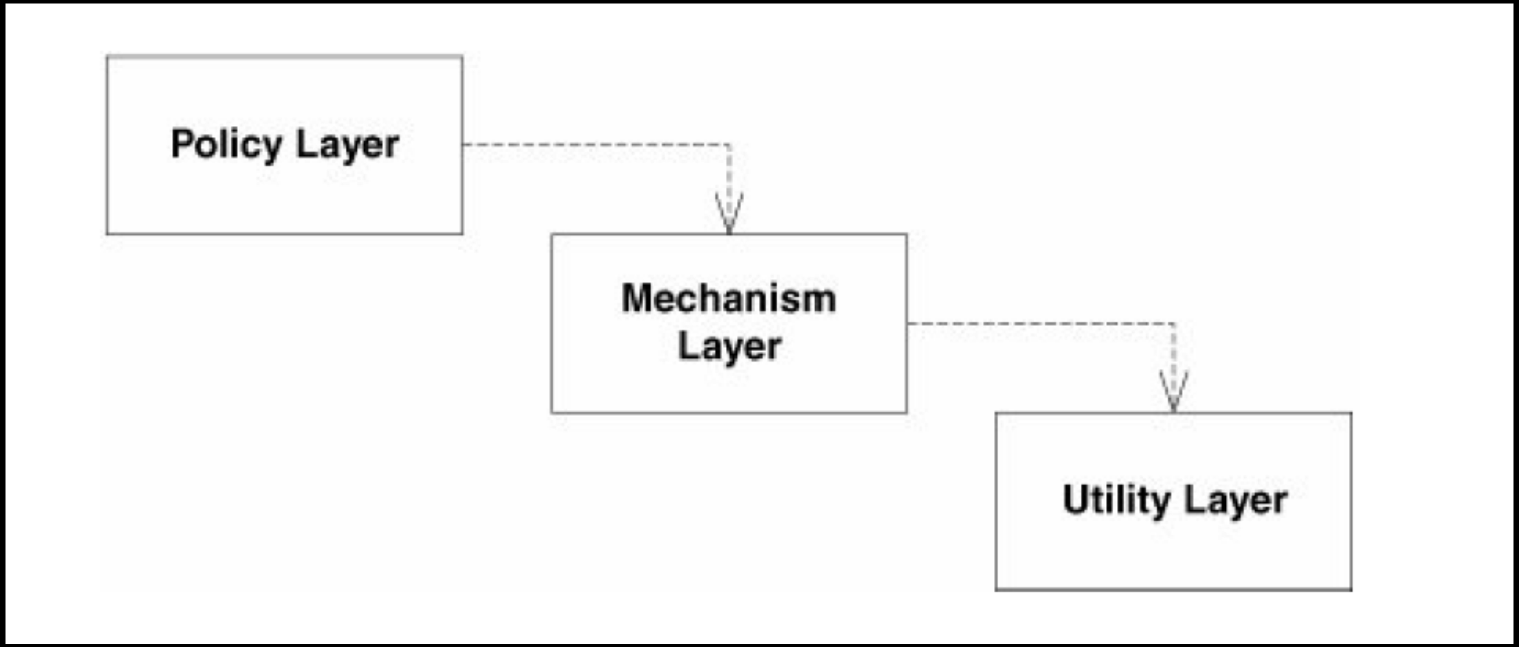
High Level vs Low Level

- A high level module defines what your application does
  - It contains the policy decisions and business models that define the entire purpose of the application
  - A high level module should be separated from and not dependent upon the implementation details implemented in lower level modules.
- Low level modules contain implementation details
  - Ex. Calls to a particular type of database

Layering in object oriented programs

- Well designed object oriented architectures should have clearly defined layers. Each layer should:
    - Provide some coherent set of services
    - Provide those services through a well defined interface

Naïve Layering:



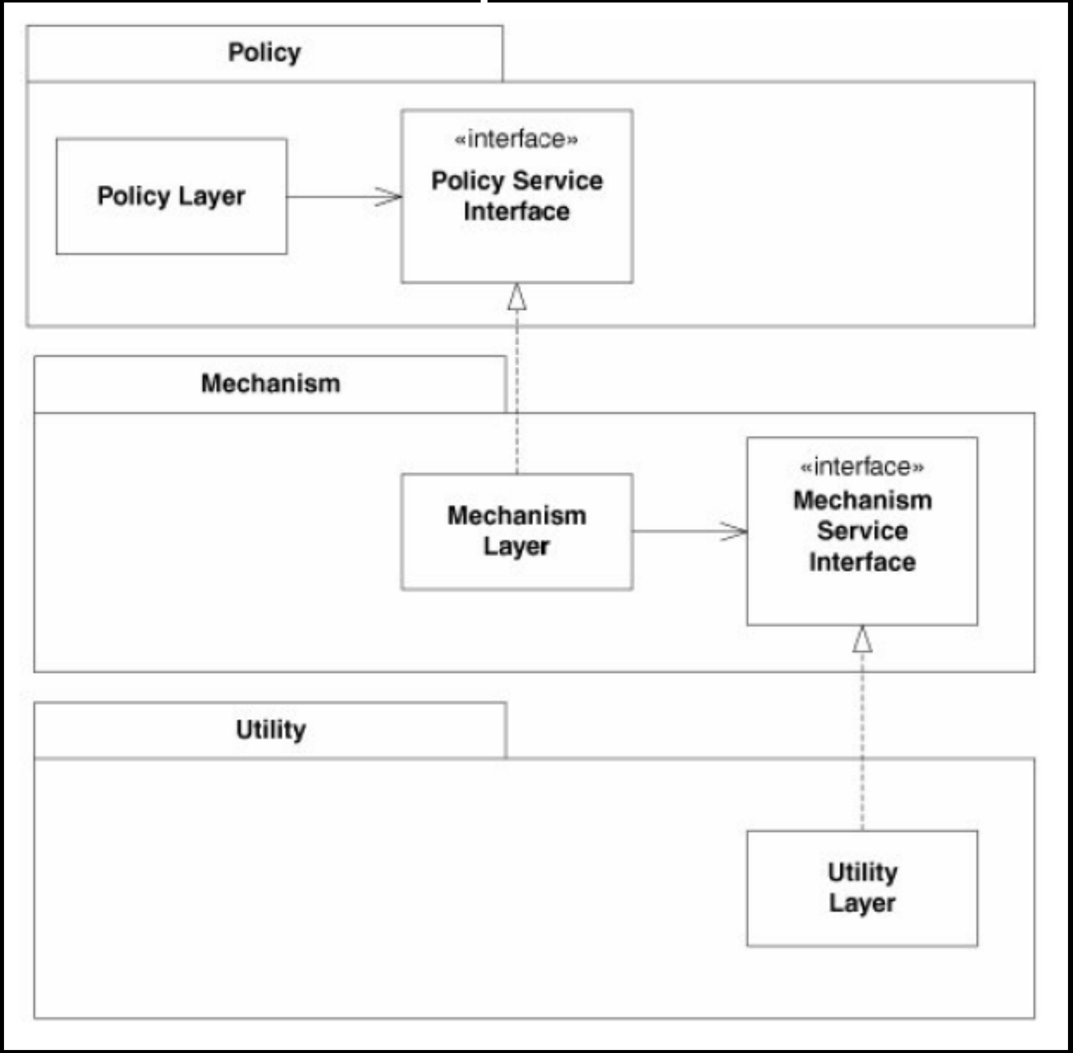
In the code:

Policy Layer calls new MechanismLayer
Mechanism Layer calls new Utility Layer

Problems?

Changes to the mechanism layer could require changes to the policy layer.
Policy Layer transitively dependent upon all layers below it

Ownership Inversion



In Code:

Policy layer does not directly declare Mechanism Layer.

No code like:
MechanismLayer ml = new MechanismLayer;

Policy layer will have something like:
IPolicyService myService;
myService can be an instance of anything that implements IPolicyService

Mechanism Layer will implement the IPolicyService interface and must conform to its expectations

Implementations dependent upon policy now!

How do I “new” up stuff?

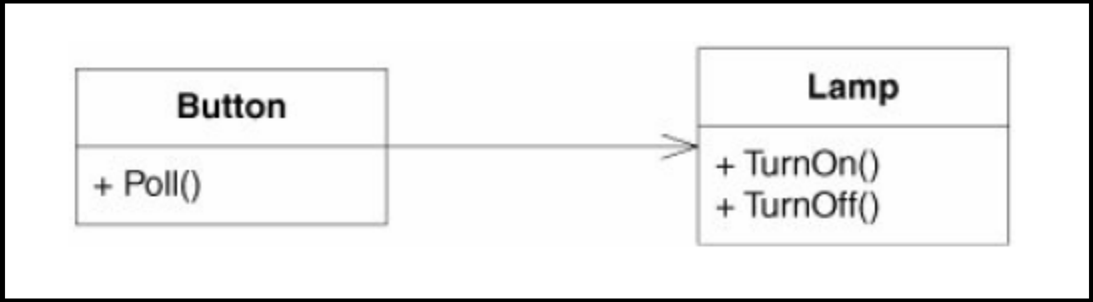
- Factory Pattern
  - Returns concrete versions of the interfaces your policy layer uses.
- Dependency Injection
  - Complicated way to inject concrete implementations of interfaces into objects.
  - Ask for ISomeInterface, inject the concrete type you have set up the injector to return.

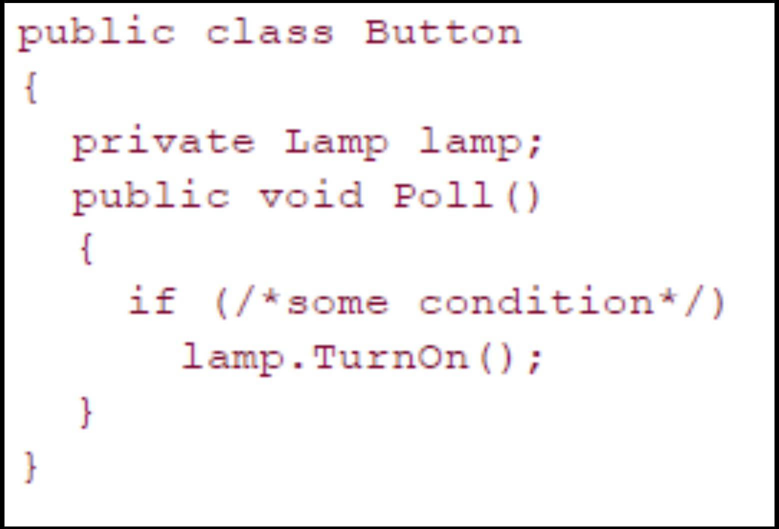
Abstractions

- No variable should have a reference to a concrete class
    - Reference to interface or abstract class means variable can be anything that implements that interface
- No class should derive from a concrete class
    - Don’t extend concrete classes
- No method should override an implemented method of any of its base classes.
    - If you extend an abstract class don’t overwrite the functionality built into it.

A Simple Example From Robert C. Martin

- The button is polling for input
  - Upon input it calls the lamp turnOn or turnOff method
- The button class depends on the lamp class, but shouldn't a button be able to turn other things on or off too?

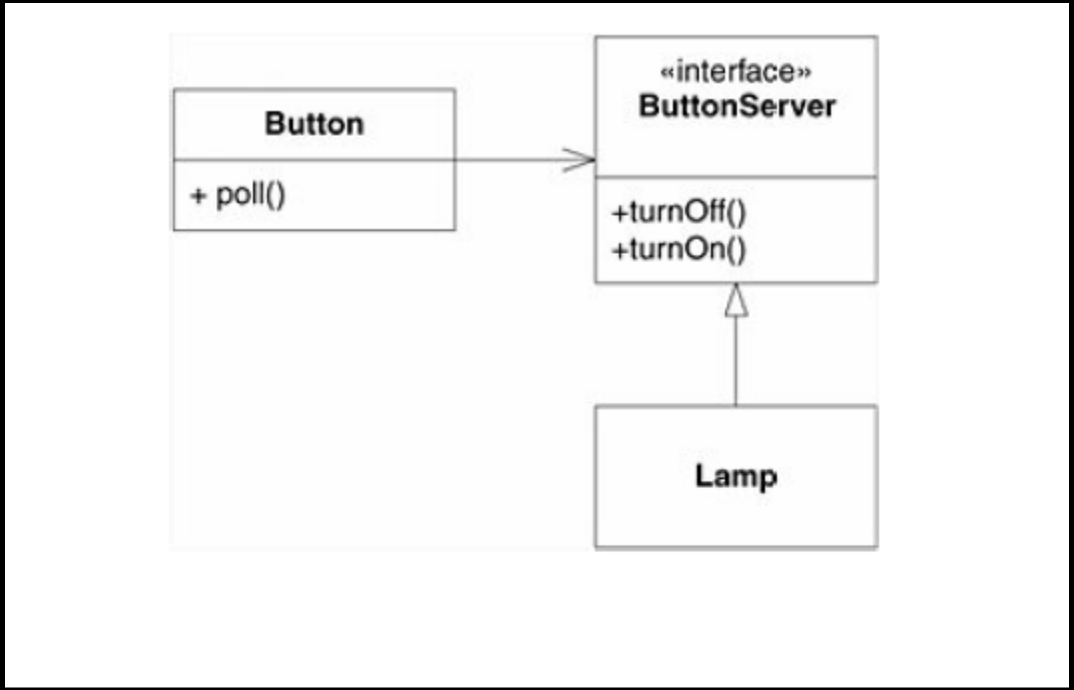




```
public class Button
{
  private Lamp lamp;
  public void Poll()
  {
    if (/*some condition*/)
      lamp.TurnOn();
  }
}
```

The Button Lamp Example With inverted dependencies

- The button no longer references a concrete class.
- The button can control anything that implements the ButtonServer interface
- Does lamp depend on button?
  - No, lamp can be controlled by anything that can control the button server interface

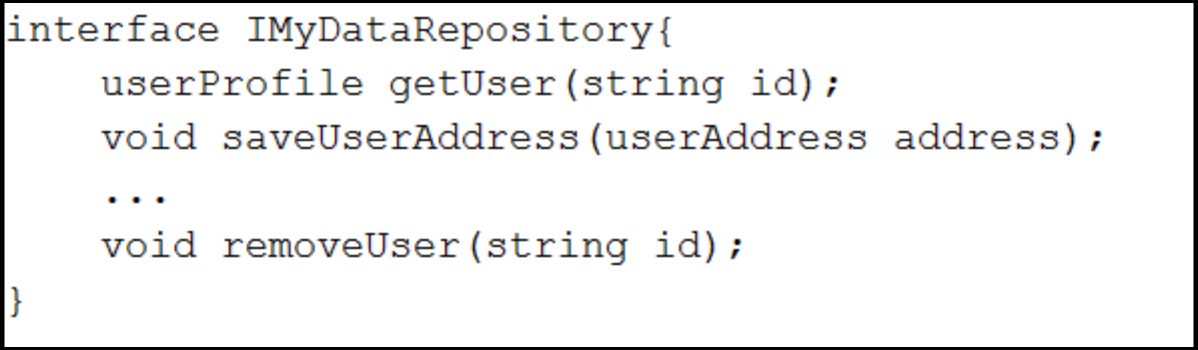


Why bother?

- Consider having your policy layer dependent upon a specific type of database
- What if your IT department doesn’t want to support the type of database you are currently using?
- What if the current database you are using is no longer adequate for your needs?
- Your entire policy layer that was based around a connection to a specific type of database is now worthless and will need to be rewritten.

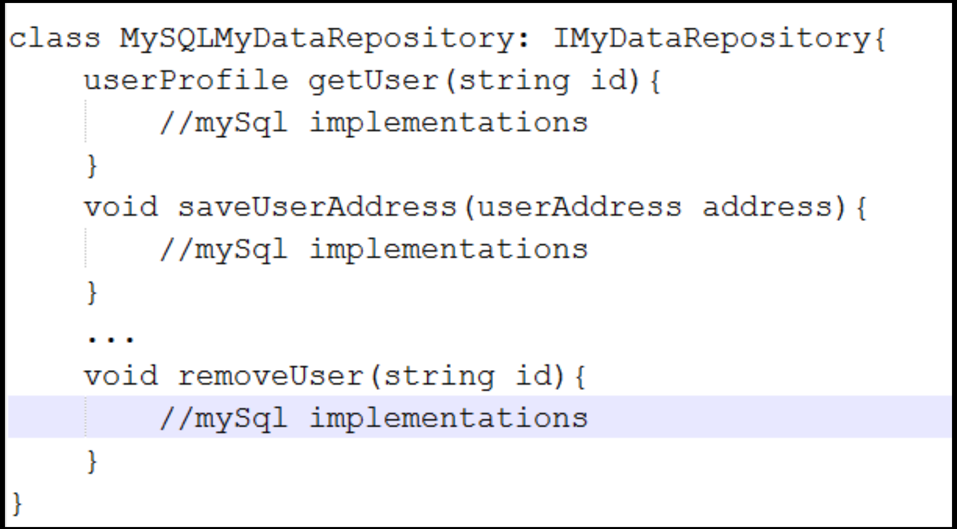
Solution: IMyDataRepository

- Your policy layer will not reference specific database objects, rather it will have a variable of type IMyDataRepository

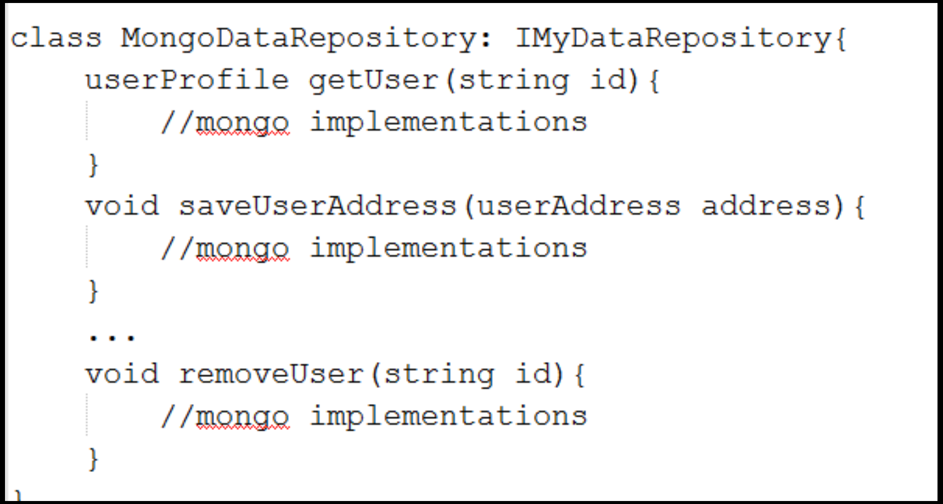


```
interface IMyDataRepository{
    userProfile getUser(string id);
    void saveUserAddress(userAddress address);
    ...
    void removeUser(string id);
}
```

Implementations of IMyDataRepository



```
class MySQLMyDataRepository: IMyDataRepository{
    userProfile getUser(string id){
        //mySql implementations
    }
    void saveUserAddress(userAddress address){
        //mySql implementations
    }
    ...
    void removeUser(string id){
        //mySql implementations
    }
}
```



```
class MongoDataRepository: IMyDataRepository{
    userProfile getUser(string id){
        //mongo implementations
    }
    void saveUserAddress(userAddress address){
        //mongo implementations
    }
    ...
    void removeUser(string id){
        //mongo implementations
    }
```