Dependency Injection

CHRISTOFFER NORING, MAHAN RAD

History

http://objectmentor.com/resources/articles/dip.pdf 1996 Bob Martin

Its the D in SOLID

S Single responsibility principle, a class should have one reason to change

O Open / closed principle , open for extension, closed for modificatiob

L Liskov substitution principle

I Interface segregation principle

D **Dependency inversion principle**

Concepts

What is a Dependency?

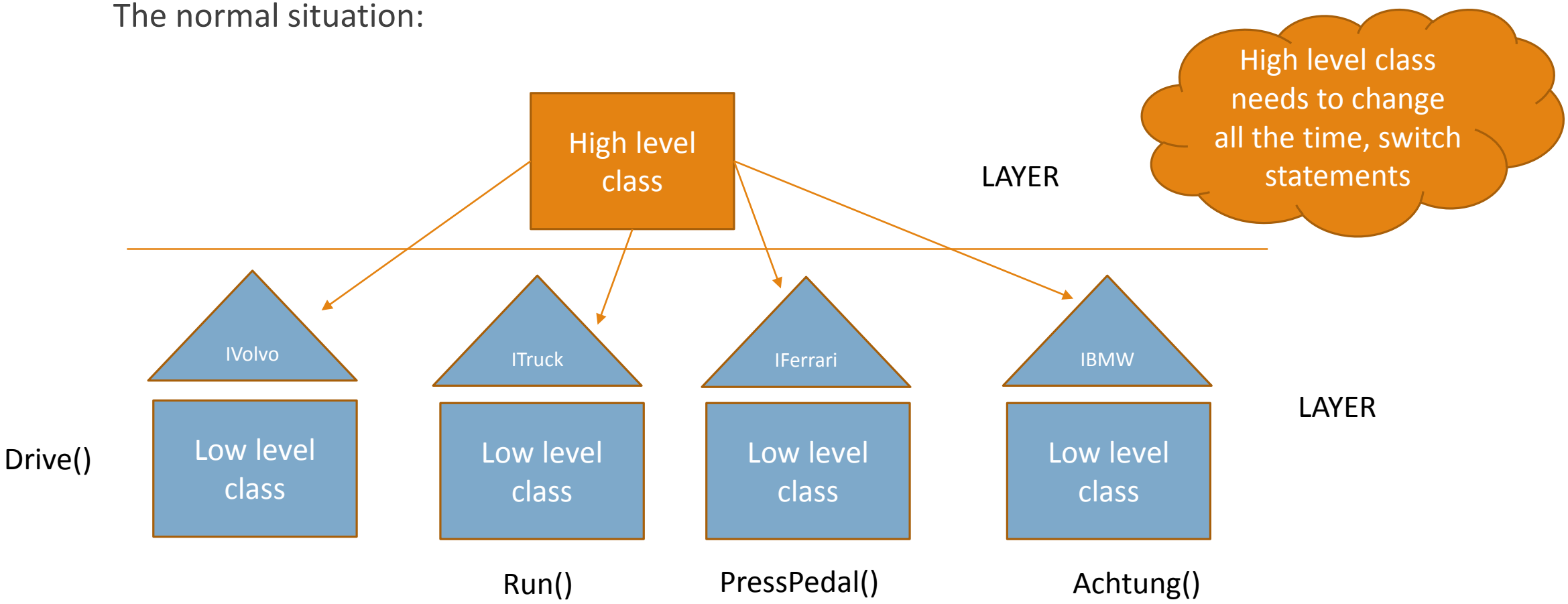
In software development terminology, a dependency is just an object that your class needs for functioning e.g: imagine a gardener wants to graft a plant. As a result, a grafting tool is a dependency in this case.

What is an abstraction

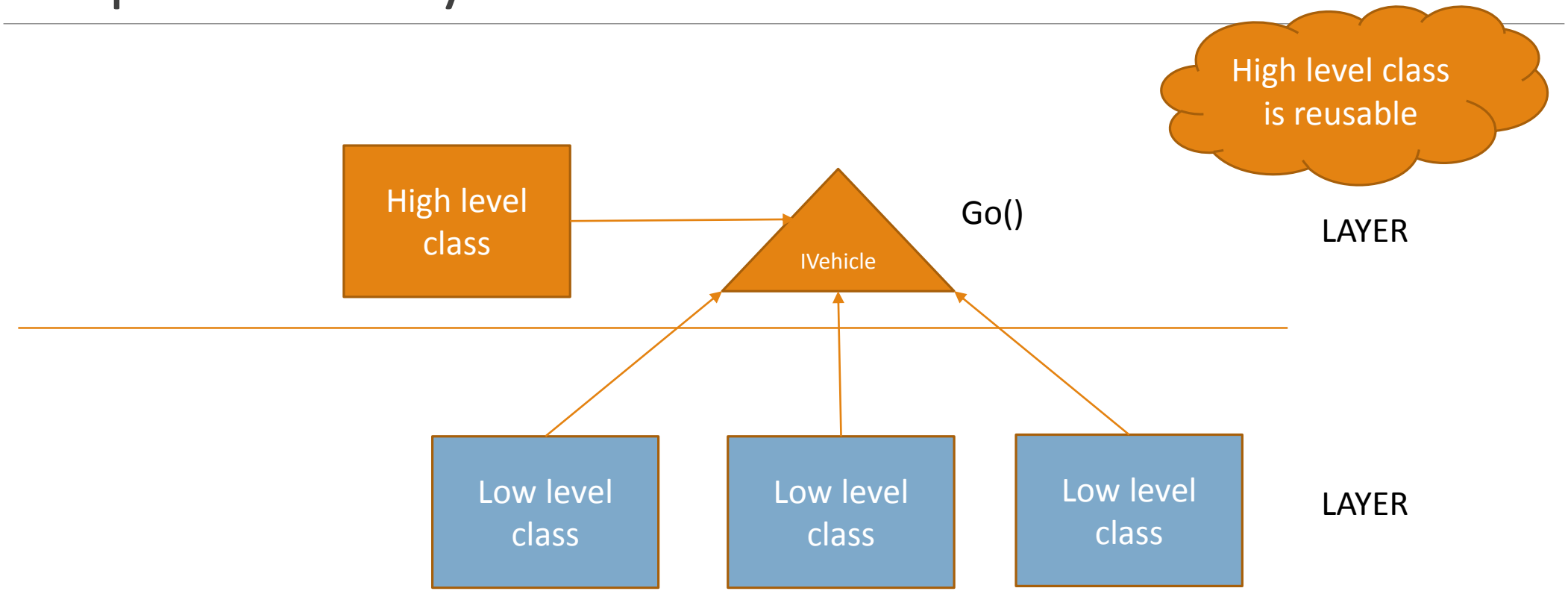
Something that represents the behavior of your class. Usually an interface or abstract class

Normal Dependency

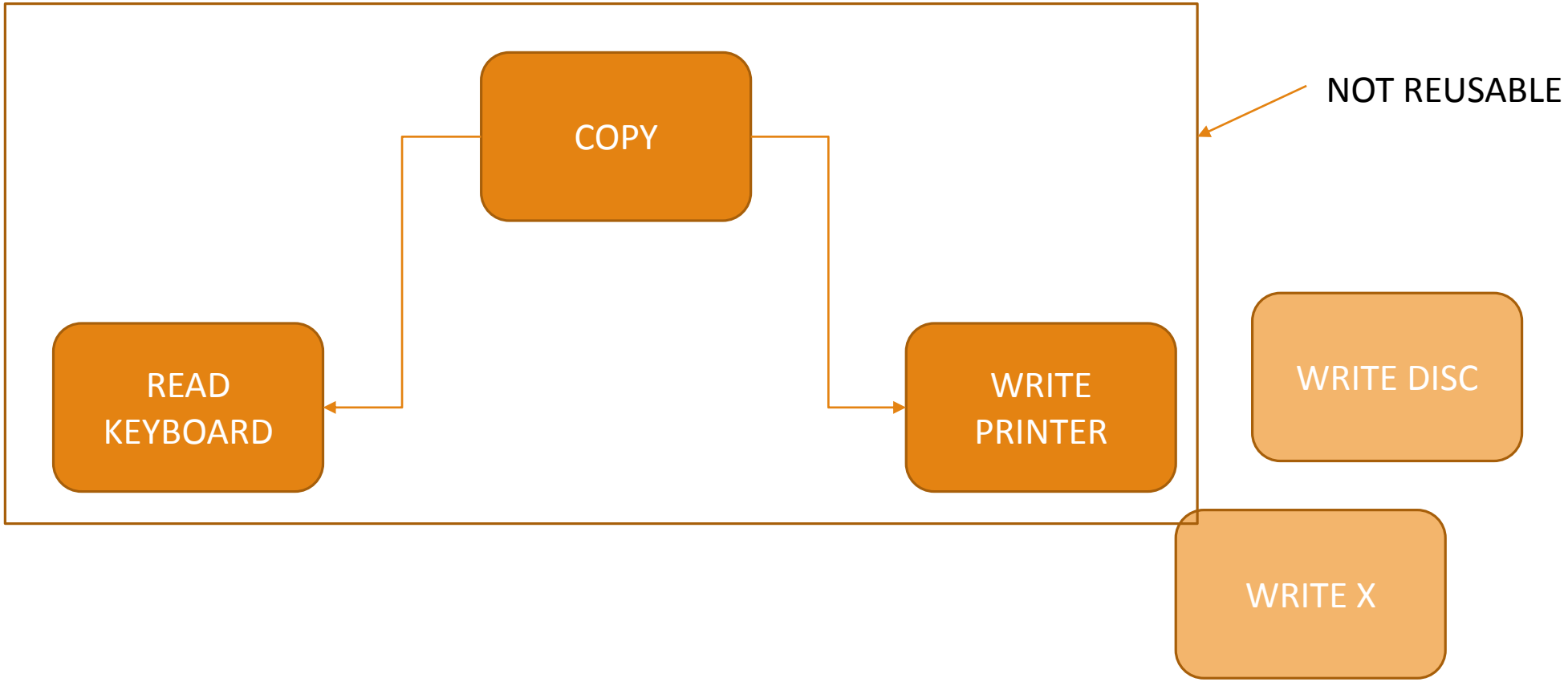
The normal situation:



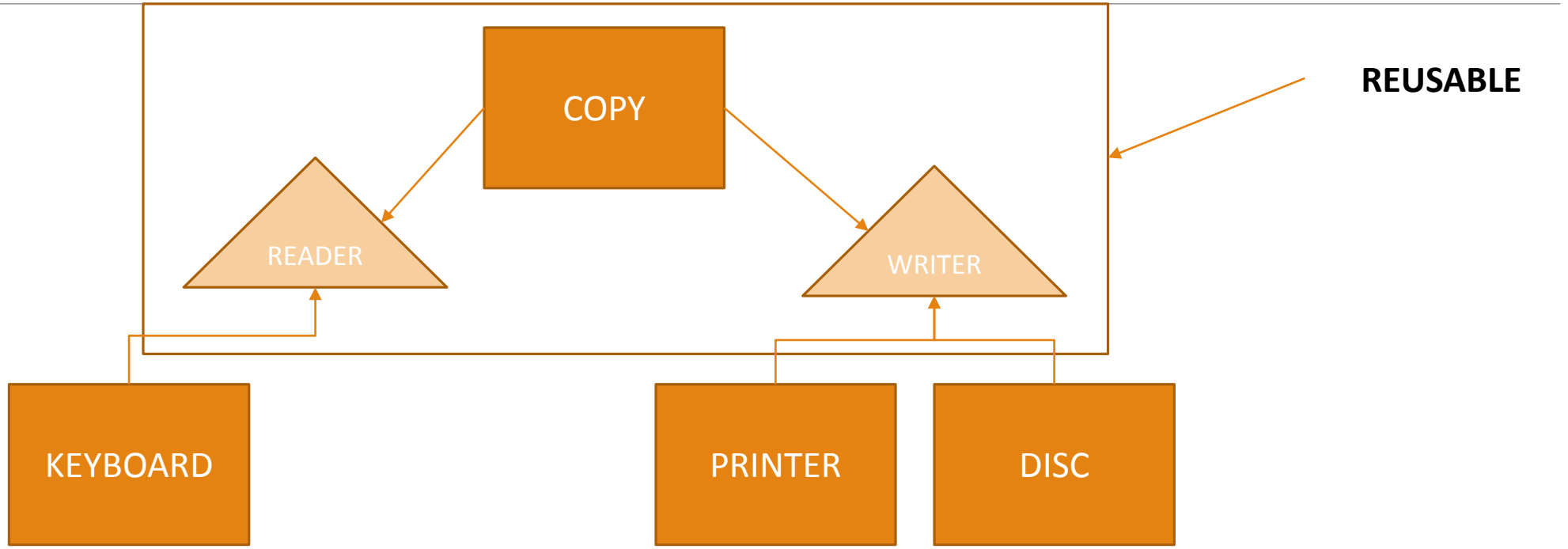
Dependency inversion



DI Example according to Uncle Bob



DI Example solution



Dependency inversion principle

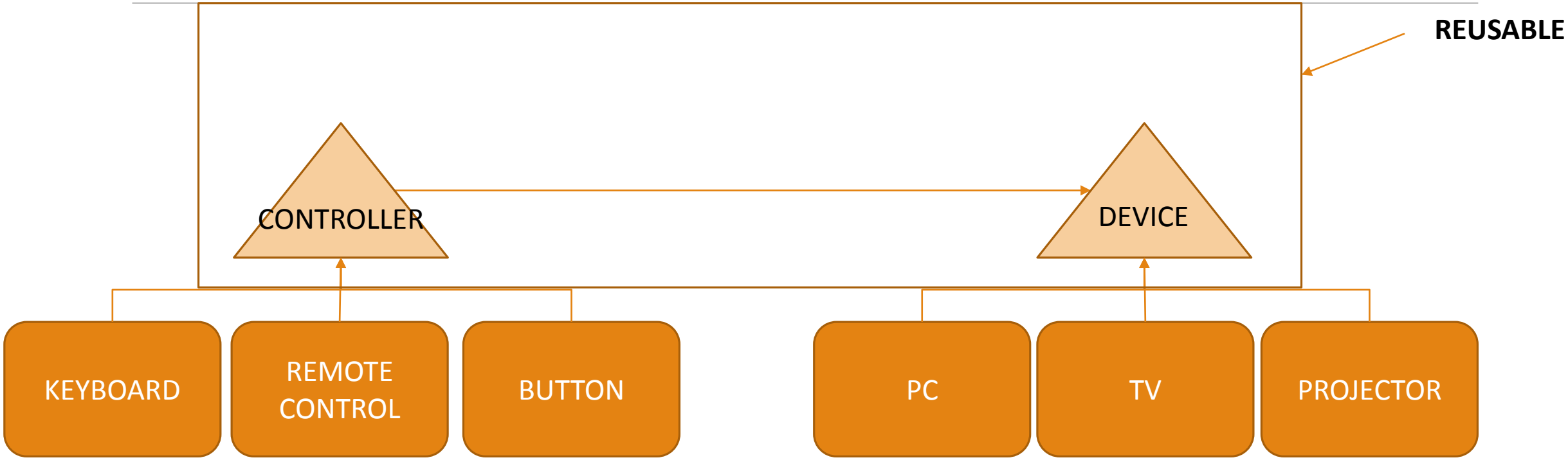
- High-level modules should not depend on low-level modules. Both should depend on abstractions

- Abstractions should not depend on details. Details should depend on abstractions.

Robert Martin

Another example

Another example - solution



Dependency injection

Dependency Injection
Well, first of all DI is a software design pattern. dependency injection is like providing the tools for the gardener by the employer. Compare this to a situation where the gardener should create his own tools to perform a simple task.

When to use it:

- Creating green field projects that are loosely coupled
- When refactoring legacy code and replace parts bit by bit

Injection
An injection is the passing of a dependency to a dependent object.

Injection types
Three types of injections **setter**, **interface** and **constructor** based.
- Separates the **creation** of a client's dependencies from its own **behavior**

Disadvantages:
- More difficult to find the implementing class, tracing
- More lines of code

Constructor injection

```
class Parser
{
   private IFilterService _filterService;

   public Parser(IFilterService filterService)
   {
       _filterService = filterService;
   }
}
```

Disadvantage: for optional dependencies, if a class uses constructor injection then **extending** it and **overriding** the constructor becomes problematic.

Advantage: if the dependency is a necessity to the class's functionality then this type of injection ensures that the dependency is provided at the time of object creation.

Setter injection

```
public void setFilter(IFilterService filterService)
{
    _filterService = filterService;
}
```

Disadvantage: the setter might be invoked multiple times and also cannot make sure if the setter has been called when logic needs the dependency (usually requires adding checks for that).

Advantage: works well with optional dependencies in a sense that you could just call the setter if you need that dependency.

Interface injection (setter injection + role interface)

Role interface

```
public interface IFilterSetter
{
   public void setFilter(IFilterService filterService);
}
```

```
class Parser : IFilterService
{
        private IFilterService _filterService;

        public Parser(IFilterService filterService)
        {
            _filterService = filterService;
        }

        public void setFilter(IFilterService filterService)
        {
            _filterService = filterService;
        }
    }
```

Example scenario : Garden

```
public class Gardener {

  private Scissors scissors;

  public void groom(Plant plant) {
      scissors = new KitchenScissors();
      // … grooming procedure that is less likely changed
  }
}

class Employer {
  public static void main(String[] args) {
      Gardener gardener = new Gardener();
      Plant roseBud = new Rose();
      gardener.groom(roseBud);
  }
}
```

Example continued

so far so good until we realize that the employer demands grooming another plant.

```
Plant baobab = new BaobabTree();
gardener.groom(baobab);
```

In order to cater for such behaviour, we only need to pass a ***“Saw”*** instead of the kitchen scissors

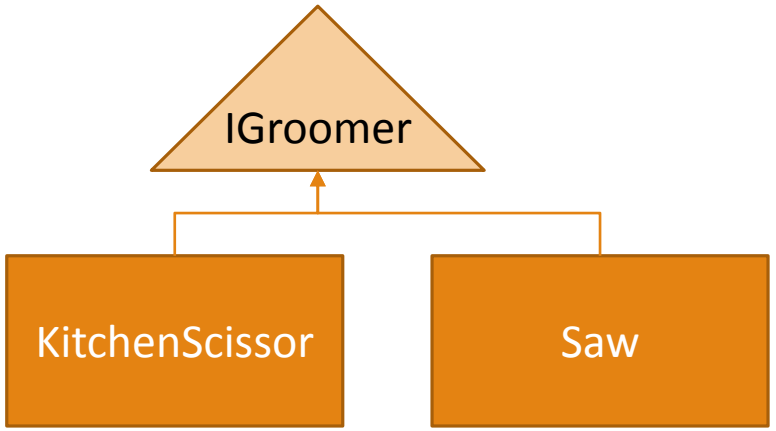
Example - solution

```
public class Gardener {

  private IGroomer groomer;

  public Gardener(IGroomer groomer) {
     this.groomer = groomer;
  }

  public void groom(Plant plant) {
      // … grooming procedure that is less likely changed
  }
}
```



Example auhentication

```
public class UserAuthenticator {

  public boolean isAuthenticated(String username, String password) {

    // unnecessary, unrelated boilerplate code used for setting up db
    Connection conn = null;
    try {
       Class.forName(driver);
       conn = DriverManager.getConnection(connectionURL);
    } catch (SQLException se)  {
       // more boilerplate code on handling probable exceptions
    }

    // even more code to fetch the password of the given user
    Statement s = conn.createStatement();
    s.executeUpdate(setProperty + requireAuth + ", 'true')");
    s.executeUpdate(setProperty + sqlAuthorization + ", 'true')");
    ResultSet rs = s.executeQuery(getProperty + requireAuth + ")");
    String realPassword = rs.next();
    // ...

    return realPassword.equals(password);
  }
}
```

Impossible to test!

Example authentication - continued

if we had the database injected as a dependency, then we could easily mock that with some... say in-memory fake database

Example authentication - solution

```
public class UserAuthenticator {

  private IUserManager userManager;

  public UserAuthenticator(IUserManager userManager) {
     this.userManager = userManager;
  }

  public boolean isAuthenticated(String username, String
password) {
      User realUser =
userManager.getUserByUsername(username);
      if (realUser == null) return false;
      return realUser.getPassword().equals(password);
  }
}
```

DI Container

What is a DI Container?
A dependency injection container (or IoC container) is a framework that automatically manages the dependencies in terms of creating, configuring, providing and destroying them in a way that the business logic will obtain the dependency exactly where and when they are needed. and in order to enjoy these benefits, … well, you have to set them up first.

DI Containers

.NET

Castle Windsor
Ninject
Spring .NET
StructureMap
AutoFac
Unity

And many more…

Java

Dagger
Guice
Spring
CDI (Java EE)
PicoContainer

Demo

Castle Windsor

Open Source IO Container

Part of the Castle Project, Active Record, MonoRail etc..

One of the first for .NET

Install-Package Castle.Windsor

Demo unity

Demo something java

Demo coupled to loose coupled

You do it. Code is at Github…

The code is coupled. Find suitable abstractions and turn them into interfaces, abstract base class and inject them.