Object Oriented Programming

- encapsulation
- inheritance
- polymorphism

Encapsulation

Encapsulation - Hiding data and requiring all interaction to be performed using exposed methods. For example, private data members and public member functions of a class.

This concept is not unique to OO and perfect encapsulation existed in C (a non-OO language). OO in the form of C++ broke perfect encapsulation, but it was partially repaired by introducing **public**, **private**, and **protected** keywords.

Java and C# weakened encapsulation further.

Many OO languages have little or no enforced encapsulation, so although encapsulation is important in OO, we may not want to say that OO depends on strong encapsulation.

Inheritance

Inheritance - Base one class/object on another while retaining a similar implementation. You can also derive a class/object from another to form a hierarchy of classes/objects.

Aka the redeclaration of variables/functions within an enclosing scope.

This was accomplished manually in C before OO came around, but it was inconvenient and multiple inheritance was difficult to achieve. OO makes this much more convenient and implicit. We could say that inheritance is a more prominent feature of OO than encapsulation, but not by much.

Polymorphism
Polymorphism - “Many forms”. An application of pointers to functions. A reference can take different forms in different instances. Different classes can be used with the same interface.

This was already possible in C prior to OO, but it was dangerous. OO makes this much safer and easier to achieve polymorphism.

Polymorphism in OO imposes discipline on indirect transfer of control. This is a key defining feature of the OO paradigm.

The Power of Polymorphism
We would like to write programs that work interchangeably with different devices. Let’s make those devices plugins to the program to achieve this.

This plugin architecture supports IO device dependence, but was not extended to other programs because of how dangerous it was to use pointers to functions.

OO removes this danger so we can use a plugin architecture anywhere, for anything.

Dependency Inversion

Before polymorphism via OOP, the flow of control was dictated by the behavior of the system, and the source code dependencies were dictated by the flow of control:

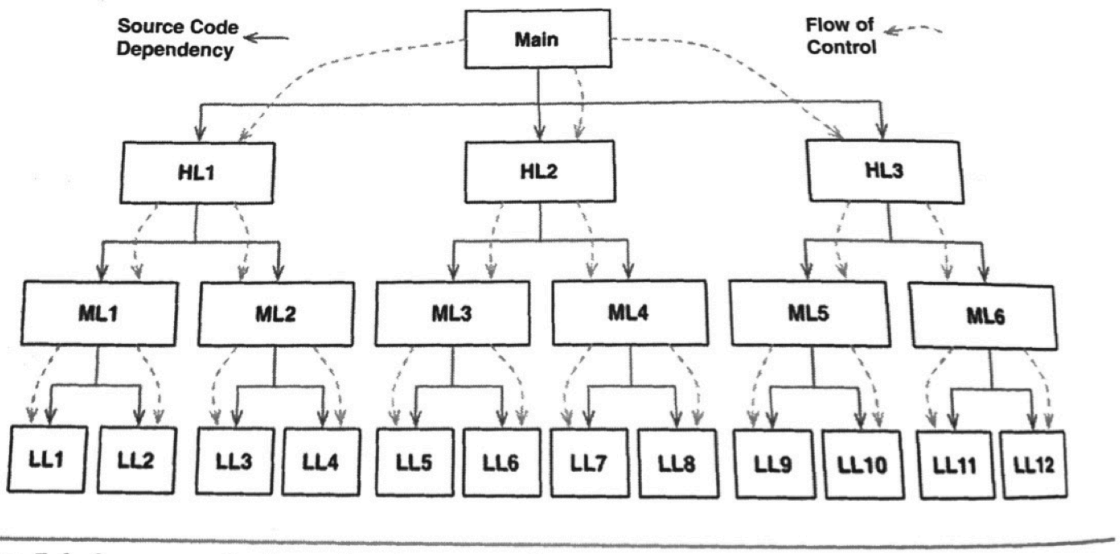

Figure 5.1 Source code dependencies versus flow of control

A source code dependency is also known as the inheritance relationship.

With polymorphism via OOP, we can invert any source code dependency, no matter where it is, by inserting an interface in between:

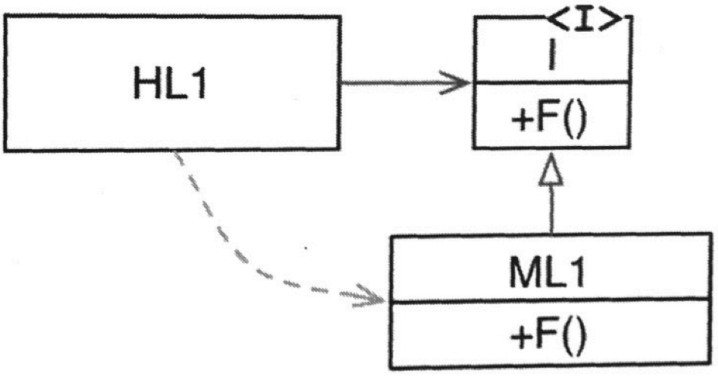

Figure 5.2 Dependency inversion

HL1 calls the F( ) function in ML1 through an interface. The flow of control is just as it was before. However, since we are using an interface, the source code dependency has been inverted.

The source code dependency points from ML1 to the interface I, which is the opposite direction of the flow of control. This is a **dependency inversion**.

The ultimate power of OO: you now have absolute control over the direction of all source code dependencies in the system!

Here is an example:

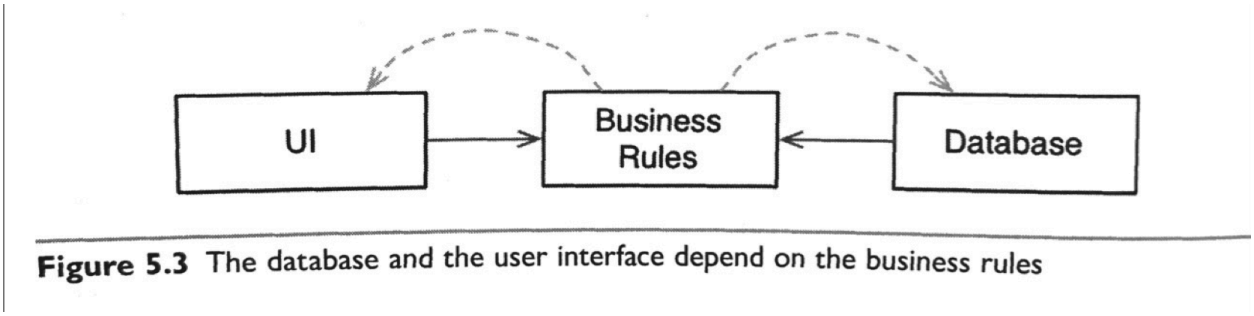

Figure 5.3 The database and the user interface depend on the business rules

The source code dependencies have been rearranged so that the UI and DB depend on the Business Logic, rather than the other way around.

The UI + DB can be plugins to the Business Logic. The Business Logic can be deployed independently of the UI + DB.

We now have **independent deployability**, which leads to **independent developability**. Different teams can work on the components.

Conclusion

OO is the ability to control every source code dependency in the system via polymorphism.

We can utilize a plugin architecture: low-level details are relegated to plugin modules. High-level are no longer dependent on low-level details.