RESUMEN FRAMEWORK

En el [desarrollo de software](/wiki/Desarrollo_de_software), un **framework** es una estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de software concretos, con base en la cual otro proyecto de [software](/wiki/Software) puede ser organizado y desarrollado. Típicamente, puede incluir soporte de [programas](/wiki/Programa_(computaci%C3%B3n)), [bibliotecas](/wiki/Biblioteca_(programaci%C3%B3n)) y un [lenguaje interpretado](/wiki/Lenguaje_interpretado) entre otros programas para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

RESUMEN INVERSION DE CONTROL

**Inversión de control** (*Inversion of Control* en inglés, **IoC**) es un método de programación en el que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones. Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones. En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

RESUMEN PRINCIPIO DE HOLLYWOOD

The Hollywood principle is a software design methodology that takes its name from the cliché response given to amateurs auditioning in Hollywood: "Don't call us, we'll call you". It is a useful paradigm that assists in the development of code with [high cohesion](/wiki/Cohesion_(computer_science)) and [low coupling](/wiki/Loose_coupling) that is easier to [debug](/wiki/Debug), maintain and test.

Most beginners are first introduced to programming from a diametrically opposed viewpoint. Programs such as [Hello World](/wiki/Hello_world_program) take control of the running environment and make calls on the underlying system to do their work. A considerable amount of successful software has been developed using the same principles, and indeed many developers need never think there is any other approach. After all, programs with linear flow are generally easy to understand.

As systems increase in complexity, the linear model becomes less maintainable. Consider for example a simple program to bounce a square around a window in your favorite operating system or window manager. The linear approach may work, up to a point. You can keep the moving and drawing code in separate procedures, but soon the logic begins to branch.

* What happens if the user resizes the window?
* Or if the square is partially off-screen?
* Are all those [system calls](/wiki/System_call) to get such resources as device contexts and interacting with the graphical user interface really part of the solution domain?

It would be much more elegant if the programmer could concentrate on the application (in this case, updating the coordinates of the box) and leave the parts common to every application to "something else".

The key to making this possible is to sacrifice the element of control. *Instead of your program running the system, the system runs your program*. In our example, our program could register for [timer events](/w/index.php?title=Timer_event&action=edit&redlink=1), and write a corresponding [event handler](/wiki/Event_handler) that updates the coordinates. The program would include other [callbacks](/wiki/Callback_(computer_science)) to respond to other [events](/wiki/Event-driven_programming), such as when the system requires part of a window to be redrawn. The system should provide suitable context information so the handler can perform the task and return. The user's program no longer includes an explicit [control path](/wiki/Control_flow), aside from initialization and registration.

[Event loop](/wiki/Event_loop) programming, however, is merely the beginning of software development following the Hollywood principle. More advanced schemes such as [event-driven object-orientation](/w/index.php?title=Event-driven_object-orientation&action=edit&redlink=1) go further along the path, by [software components](/wiki/Software_componentry) sending [messages](/wiki/Message_passing) to each other and reacting to the messages they receive. Each message handler merely has to perform its own local processing. It becomes very easy to [unit test](/wiki/Unit_testing) individual components of the system in isolation, while integration of all the components typically does not have to concern itself excessively with the dependencies between them.

[Software architecture](/wiki/Software_architecture) that encourages the Hollywood principle typically becomes more than "just" an [API](/wiki/Application_programming_interface) - instead, it may take on more dominant roles such as a [software framework](/wiki/Software_framework) or [container](/wiki/Container_(data_structure)). Examples:

* In the [Windows](/wiki/Microsoft_Windows) world:
* [MFC](/wiki/Microsoft_Foundation_Class_Library) is an example of a framework for [C++](/wiki/C%2B%2B) developers to interact with the Windows environment.
* [.NET framework](/wiki/.NET_framework) is touted as a framework for scalable [enterprise applications](/wiki/Enterprise_software).
* On the [Java](/wiki/Java_(programming_language)) side:
* [Enterprise JavaBeans](/wiki/Enterprise_JavaBean) specification describes the responsibilities of an EJB container, which must support such enterprise features as [remote procedure calls](/wiki/Remote_procedure_calls) and [transaction management](/wiki/Transaction_manager).

All of these mechanisms require some cooperation from the developer. To integrate seamlessly with the framework, the developer must produce code that follows some conventions and requirements of the framework. This may be something as simple as implementing a specific interface, or, as in the case of EJB, a significant amount of [wrapper code](/wiki/Wrapper_pattern), often produced by [code generation](/wiki/Source_code_generation) tools.

RESUMEN INYECCION DE DEPENDENCIAS

En [Informática](/wiki/Inform%C3%A1tica), **Inyección de Dependencias** (en inglés *Dependency Injection*, DI) es un [patrón de diseño](/wiki/Patr%C3%B3n_de_dise%C3%B1o) orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien cree el objeto. El término fue acuñado por primera vez por [Martin Fowler](/w/index.php?title=Martin_Fowler&action=edit&redlink=1).

RESUMEN DE MOCK OBJECT

In [object-oriented programming](/wiki/Object-oriented_programming), **mock objects** are simulated objects that mimic the behavior of real objects in controlled ways. A [computer programmer](/wiki/Computer_programming) typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a [crash test dummy](/wiki/Crash_test_dummy) to [simulate](/wiki/Simulation) the dynamic behavior of a human in vehicle impacts.

Razones para usarlo:

In a [unit test](/wiki/Unit_test), mock objects can [simulate](/wiki/Simulation) the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:

* supplies [non-deterministic](/wiki/Nondeterministic_algorithm) results (e.g. the current time or the current temperature);
* has states that are difficult to create or reproduce (e.g. a network error);
* is slow (e.g. a complete [database](/wiki/Database), which would have to be initialized before the test);
* does not yet exist or may change behavior;
* would have to include information and methods exclusively for testing purposes (and not for its actual task).

For example, an alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the test must wait until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

FRAGILE BASE CLASS

The **fragile base class problem** is a fundamental architectural problem of [object-oriented programming](/wiki/Object-oriented_programming) systems where base classes ([superclasses](/wiki/Superclass_(computer_science))) are considered "fragile" because seemingly safe modifications to a base class, when inherited by the [derived classes](/wiki/Subclass_(computer_science)), may cause the derived classes to malfunction. The programmer cannot determine whether a base class change is safe simply by examining in isolation the methods of the base class.

One possible solution is to make instance variables private to their defining class and force subclasses to use accessors to modify superclass states. A language could also make it so that subclasses can control which inherited methods are exposed publicly. These changes prevent subclasses from relying on implementation details of superclasses and allow subclasses to expose only those superclass methods that are applicable to itself.

HOOK METHOD

Reimplementacion.

In [computer programming](/wiki/Computer_programming), the term **hooking** covers a range of techniques used to alter or augment the behavior of classes or of other software components by intercepting [function calls](/wiki/Subroutine) or [messages](/wiki/Message_passing) or [events](/wiki/Event_(computing)) passed between [software components](/wiki/Module). Code that handles such intercepted function calls, events or messages is called a "hook".