Inversion of Control ( IoC)

In [software engineering](http://en.wikipedia.org/wiki/Software_engineering), **Inversion of Control** (**IoC**) is an [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming) practice whereby the object coupling is bound at [run time](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)) by an "assembler" object and is typically not knowable at [compile time](http://en.wikipedia.org/wiki/Compile_time) using [static analysis](http://en.wikipedia.org/wiki/Static_code_analysis).

In traditional programming, the [flow](http://en.wikipedia.org/wiki/Control_flow) of the [business logic](http://en.wikipedia.org/wiki/Business_logic) is determined by objects that are statically assigned to one another. With **Inversion of Control**, the flow depends on the object graph that is instantiated by the assembler and is made possible by object interactions being defined through abstractions. The binding process is achieved through [dependency injection](http://en.wikipedia.org/wiki/Dependency_injection), although some argue that the use of a [service locator](http://en.wikipedia.org/wiki/Service_locator_pattern) also provides Inversion of Control.

In order for the assembler to bind objects to one another, the objects must possess compatible abstractions. For example, class A may delegate behavior to interface I which is implemented by class B; the assembler instantiates A and B then injects B to A.

1. Delegates behavior to Interface I
2. Implements I

Assembler instantiates A, B then injects B to A

In practice, Inversion of Control is a style of software construction where reusable code controls the execution of problem-specific code. It carries the strong connotation that the reusable code and the problem-specific code are developed independently, which often results in a single integrated application. Inversion of Control as a design guideline serves the following purposes:

* There is a [decoupling](http://en.wikipedia.org/wiki/Object-oriented_programming#Decoupling) of the execution of a certain task from implementation.
* Every module can focus on what it is designed for.
* Modules make no assumptions about what other systems do but rely on their [contracts](http://en.wikipedia.org/wiki/Design_by_contract).
* Replacing modules has no [side effect](http://en.wikipedia.org/wiki/Side_effect_(computer_science)) on other modules.

**Dependency injection** is a [software design pattern](http://en.wikipedia.org/wiki/Software_design_pattern) that allows a choice of component to be made at run-time rather than compile time. This can be used, for example, as a simple way to load [plugins](http://en.wikipedia.org/wiki/Plug-in_(computing)) dynamically or to choose [mock objects](http://en.wikipedia.org/wiki/Mock_object) in test environments vs. real objects in production environments

More important than the applied technique, however, is the optimization of the purposes.

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Inversion_of_control&action=edit&section=3)**] Examples**

public class ServerFacade {

public <K, V> V respondToRequest(K request) {

if (businessLayer.validateRequest(request)) {

DAO.getData(request);

return Aspect.convertData(request);

}

return null;

}

}

This basic outline in Java gives an example of code following the IoC methodology. It is important, however, that in the ServerFacade a lot of assumptions are made about the data returned by the [data access object](http://en.wikipedia.org/wiki/Data_access_object) (DAO).

Although all these assumptions might be valid at some time, they couple the implementation of the ServerFacade to the DAO implementation. Designing the application in the manner of Inversion of Control would hand over the control completely to the DAO object. The code would then become

public class ServerFacade {

public <K, V> V respondToRequest(K request, DAO dao) {

return dao.getData(request);

}

}

The example shows that the way the method respondToRequest is constructed determines if IoC is used. It is the way that parameters are used that define IoC. This resembles the [message-passing](http://en.wikipedia.org/wiki/Message_Passing) style that some object-oriented programming languages have been using.

### APIs that use inversion of control

[SAX](http://en.wikipedia.org/wiki/Simple_API_for_XML) is an example of an API that uses inversion of control throughout (after initialisation). SAX is generally more efficient than [DOM](http://en.wikipedia.org/wiki/Document_Object_Model), but DOM is often considered more convenient to program with, because it is not necessary to deal with inversion of control.[[](http://en.wikipedia.org/wiki/Inversion_of_control#cite_note-devx-6)

**Benefits of Dependency Injection**

One benefit of using the dependency injection approach is the reduction of [boilerplate code](http://en.wikipedia.org/wiki/Boilerplate_code) in the application objects since all work to initialize or set up dependencies is handled by a provider component.[[1]](http://en.wikipedia.org/wiki/Dependency_Injection#cite_note-0)

Another benefit is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This is useful in [unit testing](http://en.wikipedia.org/wiki/Unit_testing), as it is easy to inject a [fake implementation](http://en.wikipedia.org/wiki/Mock_object) of a service into the object being tested by changing the configuration file, or overriding component registrations at run-time.

Furthermore, dependency injection facilitates the writing of testable code.

**Some DI Frameworks**

There are a number of frameworks available today to help the developer. The ones I have found useful are :

* [Spring Framework](http://springframework.org/) : A substantially large framework which offers a number of other capabilities apart from Dependency Injection.
* [PicoContainer](http://picocontainer.codehaus.org/) : A fairly small tightly focused DI container framework.
* [HiveMind](http://jakarta.apache.org/hivemind/) : Another DI container framework.
* [XWork](http://www.opensymphony.com/xwork/) : Primarily a command pattern framework which very effectively leverages Dependency Injection. While it is an independent framework in its own right, it is often used in conjunction with [Webwork](http://www.opensymphony.com/webwork/)

More complicated implementations of Dependency Injection, such as [Spring](http://en.wikipedia.org/wiki/Spring_Framework), [Google Guice](http://en.wikipedia.org/wiki/Guice), and [Microsoft Managed Extensibility Framework (MEF)](http://en.wikipedia.org/wiki/Managed_Extensibility_Framework), automate this procedure. These frameworks identify constructor arguments or properties on the objects being created as requests for dependent objects, and automatically inject constructor arguments or set properties with pre-constructed instances of dependencies as part of the process of creating the dependent object. The client makes a request to the dependency injection system for an implementation of a particular interface; the dependency injection system creates the object, automatically filling in dependencies as required.

### Highly coupled dependency

The following example shows code with no dependency injection applied:

public class VerySimpleStockTraderImpl implements IAutomatedStockTrader {

private IStockAnalysisService analysisService = new StockAnalysisServiceImpl();

private IOnlineBrokerageService brokerageService = new NewYorkStockExchangeBrokerageServiceImpl();

public void executeTrades() {

for (String stockSymbol : brokerageService.getStockSymbols()) {

double askPrice = brokerageService.getAskingPrice(stockSymbol);

double estimatedValue = analysisService.getEstimatedValue(stockSymbol);

if (askPrice < estimatedValue) {

brokerageService.putBuyOrder(stockSymbol, 100, askPrice);

}

}

}

}

public class MyApplication {

public static void main(String[] args) {

IAutomatedStockTrader stockTrader = new VerySimpleStockTraderImpl();

stockTrader.executeTrades();

}

}

The VerySimpleStockTraderImpl class creates instances of the IStockAnalysisService, and IOnlineBrokerageService by hard-coding constructor references to the concrete classes that implement those services.

### [[edit](http://en.wikipedia.org/w/index.php?title=Dependency_injection&action=edit&section=6)] Manually injected dependency

Refactoring the above example to use manual injection:

public class VerySimpleStockTraderImpl implements IAutomatedStockTrader {

private IStockAnalysisService analysisService;

private IOnlineBrokerageService brokerageService;

public VerySimpleStockTraderImpl(

IStockAnalysisService analysisService,

IOnlineBrokerageService brokerageService) {

this.analysisService = analysisService;

this.brokerageService = brokerageService;

}

public void executeTrades() {

…

}

}

public class MyApplication {

public static void main(String[] args) {

IStockAnalysisService analysisService = new StockAnalysisServiceImpl();

IOnlineBrokerageService brokerageService = new NewYorkStockExchangeBrokerageServiceImpl();

IAutomatedStockTrader stockTrader = new VerySimpleStockTraderImpl(

analysisService,

brokerageService);

stockTrader.executeTrades();

}

}

In this example, MyApplication.main plays the role of dependency injector, selecting the concrete implementations of the dependencies required by VerySimpleStockTraderImpl, and supplying those dependencies via **constructor injection**.