The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor {

private SpellChecker checker;

public TextEditor() {

this.checker = new SpellChecker();

}

}

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

public class TextEditor {

private IocSpellChecker checker;

public TextEditor(IocSpellChecker checker) {

this.checker = checker;

}

}

In the first code example we are instantiating SpellChecker (this.checker = new SpellChecker();), which means the TextEditor class directly depends on the SpellChecker class.

In the second code example we are creating an abstraction by having the SpellChecker dependency class in TextEditor's constructor signature (not initializing dependency in class). This allows us to call the dependency then pass it to the TextEditor class like so:

SpellChecker sc = new SpellChecker(); // dependency

TextEditor textEditor = new TextEditor(sc);

Now the client creating the TextEditor class has control over which SpellChecker implementation to use because we're injecting the dependency into the TextEditor signature.

If you follow these simple two steps, you have done inversion of control:

1. Separate **what**-to-do part from **when**-to-do part.
2. Ensure that **when** part knows as *little* as possible about **what** part; and vice versa.

There are several techniques possible for each of these steps based on the technology/language you are using for your implementation.

--

The *inversion* part of the Inversion of Control (IoC) is the confusing thing; because *inversion* is the relative term. The best way to understand IoC is to forget about that word!

--

Examples

* Event Handling. Event Handlers (what-to-do part) -- Raising Events (when-to-do part)
* Dependency Injection. Code that constructs a dependency (what-to-do part) -- instantiating and injecting that dependency for the clients when needed, which is usually taken care of by the DI tools such as Dagger (when-to-do-part).
* Interfaces. Component client (when-to-do part) -- Component Interface implementation (what-to-do part)
* xUnit fixture. Setup and TearDown (what-to-do part) -- xUnit frameworks calls to Setup at the beginning and TearDown at the end (when-to-do part)
* Template method design pattern. template method when-to-do part -- primitive subclass implementation what-to-do part
* DLL container methods in COM. DllMain, DllCanUnload, etc (what-to-do part) -- COM/OS (when-to-do part)