InversionOfControl

Martin Fowler
26 June 2005

Inversion of Control is a common phenomenon that you come across when extending frameworks. Indeed it's often seen as a defining characteristic of a framework.

Let's consider a simple example. Imagine I'm writing a program to get some information from a user and I'm using a command line enquiry. I might do it something like this

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

In this interaction, my code is in control: it decides when to ask questions, when to read responses, and when to process those results.

However if I were were to use a windowing system to do something like this, I would do it by configuring a window.

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

There's a big difference now in the flow of control between these programs - in particular the control of when the `process_name` and `process_quest` methods are called. In the command line form I control when these methods are called, but in the window example I don't. Instead I hand control over to the windowing system (with the `Tk.mainloop` command). It then decides when to call my methods, based on the bindings I made when creating the form. The control is inverted - it calls me rather me calling the framework. This phenomenon is Inversion of Control (also known as the Hollywood Principle - "Don't call us, we'll call you").

> *One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework*

Find similar articles at these tags

application architecture

API design

object collaboration design

> *for a particular application.*
>
> -- *Ralph Johnson and Brian Foote*

Inversion of Control is a key part of what makes a framework different to a library. A library is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.

A framework embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

There are various ways you can plug your code in to be called. In the ruby example above, we invoke a bind method on the text entry field that passes an event name and a Lambda as an argument. Whenever the text entry box detects the event, it calls the code in the closure. Using closures like this is very convenient, but many languages don't support them.

Another way to do this is to have the framework define events and have the client code subscribe to these events. .NET is a good example of a platform that has language features to allow people to declare events on widgets. You can then bind a method to the event by using a delegate.

The above approaches (they are really the same) work well for single cases, but sometimes you want to combine several required method calls in a single unit of extension. In this case the framework can define an interface that a client code must implement for the relevant calls.

EJBs are a good example of this style of inversion of control. When you develop a session bean, you can implement various methods that are called by the EJB container at various lifecyle points. For example the Session Bean interface defines `ejbRemove`, `ejbPassivate` (stored to secondary storage), and `ejbActivate` (restored from passive state). You don't get to control when these methods are called, just what they do. The container calls us, we don't call it.

These are complicated cases of inversion of control, but you run into this effect in much simpler situations. A template method is a good example: the super-class defines the flow of control, subclasses extend this overriding methods or implementing abstract methods to do the extension. So in JUnit, the framework code calls `setUp` and `tearDown` methods for you to create and clean up your text fixture. It does the calling, your code reacts - so again control is inverted.

There is some confusion these days over the meaning of inversion of control due to the rise of IoC containers; some people confuse the general principle here with the specific styles of inversion of control (such as dependency injection) that these containers use. The name is somewhat confusing (and ironic) since IoC containers are generally regarded as a competitor to EJB, yet EJB uses inversion of control just as much (if not more).

Etymology: As far as I can tell, the term Inversion of Control first came to light in Johnson and Foote's paper Designing Reusable Classes, published by the Journal of Object-Oriented Programming in 1988. The paper is one of those that's aged well - it's well worth a read now over fifteen years later. They think they got the term from somewhere else, but can't remember what. The term then insinuated itself into the object-oriented community and appears in the Gang of Four book. The more colorful synonym 'Hollywood Principle' seems to originate in a paper by Richard Sweet on Mesa in 1983. In a list of design goals he writes: *"Don't call us, we'll call you (Hollywood's*

Law): A tool should arrange for Tajo to notify it when the user wishes to communicate some event to the tool, rather than adopt an 'ask the user for a command and execute it' model." John Vlissides wrote a column for C++ report that provides a good explanation of the concept under the 'Hollywood Principle' moniker. (Thanks to Brian Foote and Ralph Johnson for helping me with the Etymology.)

Guides

Intro
Design
Agile
NoSQL
DSL
Delivery
About
Me

Popular Articles

New Methodology
Dependency Injection
Mocks aren't Stubs
Is Design Dead?
Continuous Integration

Books

NoSQL Distilled
Domain-Specific Languages
Refactoring
Patterns of Enterprise Application Architecture
UML Distilled
Analysis Patterns
Planning Extreme Programming
Signature Series

Site Sections

FAQ
Content Index
Bliki
Books
DSL Catalog
EAA Catalog
EAA Dev
Photos

ThoughtWorks

Blogs
Careers
Mingle
Twist
Go