InversionOfControl

Inversion of Control: An Introduction

Inversion of Control (IoC) has become something of a buzzword. The problem with buzzwords is that they are often not very well understood. The purpose of this article is to explore the concept of IoC, its (many) definitions, and look at what are sometimes called "IoC Frameworks."

Design Patterns

Before we discuss Inversion of Control, we first need to discuss **Design Patterns.** Design Patterns are programming "best practices" – common concepts which can be reused in a myriad of solutions:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. – Christopher Alexander

Examples of design patterns includes the Observer pattern, the Singleton pattern, and Model View Controller (MVC) architecture. Interestingly enough, Inversion of Control *isn't* really a design pattern (see Stefano's thoughts), but more of a general principle or set of design patterns which include:

- Separation of Concerns
- Interface Driven Design
- Dependency Management (via Dependency Injection or Service Lookup)
- Lifecycle Management
- Component Oriented Programming

SeparationOfConcerns (SoC) is a very important design pattern which is central to IoC development. In fact, without first applying proper SoC principles, it can be very difficult to take advantage of IoC. Separation of Concerns means a problem should be seperated into a set of components each of which focus on only one core concern. Imagine, for example, a large corporation made of many divisions such as human resources, accounting, logistics, sales, market, manufacturing, etc. Each of these divisions handles only one aspect, or concern, of the overall business. The human resources team does not investigate product defects and the marketing team does administer the corporate intranet. Likewise, a software application will generally have many concerns – logging, caching, security, business logic, persistance, etc. Separating out these concerns into seperate software modules allows each module to concern itself with only doing one job well.

(fixme: more discussion of the other design patterns)

What Are We Trying To Invert?

So, what is Inversion of Control? Well, let's look at some of the problems IoC tries to solve:

- Object and Component Coupling
- Implementation Lock
- Unpredictable Lifecycles
- Code rewriting (instead of reuse)
- Scattered Configuration
- Insecure Code
- Complexity

The goal of IoC is to isolate control. Inversion of Control originated as *The Hollywood Principle*: "don't call us, we'll call you." Under this premise, objects should not take control, but expect to be controlled and have everything handed to them on a silver platter. Well, maybe not a silver platter, but you'll get the idea.

There are many types of control which can be inverted. In general, IoC frameworks are interested in inverting:

- Dependency Control: How do we get a handle on other objects/components?
- Resource Control: How do we get access to system resources such as the file system or sockets?
- Configuration Control: How does a component get configuration or parameter information?
- Lifecycle Control: When does an object start or stop and who starts or stops it?
- and so on...

In many applications, objects assume this control themselves. Not only does this mean that each object or component has a lot of work to do, but it also takes a large amount of coordination to ensure that things are in control and not out of it. By centralizing application control into a single component called a container (see below), applications become more stable.

Where Does the Control Go?

We need to get some terminology down.

Component :

Container :

(we'll save the rest for later)

Types of Inversion of Control

IoC Applied

Related Resources

- IoC Container Internals by Leo Simons