The Rich Engineering Heritage Behind Dependency Injection

(or “Why dependency injection is here to stay, what it represents, and how to make it more effective and useful.”)

Author: Andrew McVeigh

Background: Andrew has worked as a software developer and designer for 18 years. A dominant theme throughout his career has been object orientation – the application of OO modeling, design and implementation. Initially a C++ and Smalltalk programmer, he has been doing Java development for around eight years. In addition to ongoing commercial development work in investment banking, Andrew is finishing off a PhD in software engineering at Imperial College.

Introduction

Dependency injection (DI) is now well established as a sensible and practical way to “wire” up Java programs. I'm talking particularly of the XML configuration approach used by Spring. Spring has led the way to mainstream acceptance of DI, particularly for Java server side architectures. DI has proven its worth as a way of making the components of a system and the architecture explicit. It's also very useful as a way of unit testing components.

So what is the engineering heritage behind DI? Did it just appear out of nowhere? Is it a passing fad, something to be replaced when the hype cycle recedes, a reaction against the horrors of EJB JNDI overuse? Or does it have a sound basis in software engineering theory? If so, what can we learn from this?

Well, to academics familiar with the research behind software components over the last twenty years, it is clear that DI is very closely related to a well-established area of software research known as architecture description languages (ADLs). The job of an ADL is to wire components together via configuration.

In fact, the Spring bean configuration configuration language is (technically speaking) a non-hierarchical ADL. In this article, I'll briefly compare Spring configuration and some of the older approaches, showing how similar they are. I'll also show how ADLs developed, giving a unique insight into possible future developments in this area.

The bottom line is that Spring DI represents mainstream acceptance of the component approach used by ADLs. DI is here to stay, and will get more important as it is applied to larger systems, more varied domains, and applied at a finer grained level. This is because it represents a principled and academically sound way to form software out of components. In addition, the fact that DI was largely developed independently of any ADL research provides powerful practical validation of the concepts from both a practical and academic point of view.

Mandatory Disclaimer

I do research into software components as part of my PhD, having created my own experimental

ADL called Backbone, which focusses on representing system evolution. My supervisors created Darwin, one of the better known ADLs. As such, this is an intentionally biased view towards the academic literature.

I've also worked with Rod Johnson on a previous commercial project, although I have no affiliation with SpringSource and I haven't contributed to Spring in any way either conceptually or via code. I currently use Spring.Net on a commercial project.

Software Components

What is a software component? This can be a seriously hard question to answer, and if you ask ten different developers, you are likely to get at least ten different answers!

The intuition behind software components, however, is quite easy to pin down: making software should be like wiring together electronics components. Components should be interchangeable, tangible units with well defined provided and required interfaces. In short, we want to wire up components together to make a software system. We also want to take existing components out of a system, and wire in new ones as long as their interfaces are compatible.

This is the intuition that many people get from reading possibly one of the most influential papers on the subject – Professor Doug McIlroy's address on Mass Produced Software Components, delivered to the NATO software engineering conference. This address was given in 1968!

Let me then offer a simple definition of a software component distilled out of the history of ADLs:

> *A component is a unit of software that can be instantiated, and is insulated from its environment by explicitly indicating which services (via interfaces) are provided and required.*

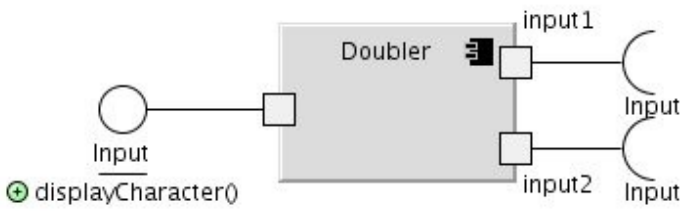
This is a surprisingly simple definition, but it is also profound, and it has taken a long time for people to achieve any level of consensus on this. A Java class definition generally satisfies the "provided" part of the equation (via interface implementation), but fails as a component because it doesn't have to explicitly indicate what interfaces it requires from other components. It could depend on concrete classes, or all manner of other things.

Interfaces are really the key here. They allow components to be substituted as long as the interfaces match up.

Further, unlike a module, a component can be instantiated. In general, most ADLs allow components to be instantiated as many times as required, like how class instantiation works.

So that's really it at the Java level. A Java class can be used as a component as long as there is some convention or restriction which constraints it to be explicit about what interfaces it requires. This is how Spring bean configuration works, and the mainstream acceptance of this technique represents an important milestone which should not be underestimated.

Below is the UML representation of a Doubler component that receives a character from a method on the provided Input interface, and sends it to its two required interfaces. Next to the UML picture is the equivalent Java code, which perhaps shows the situation more clearly. The "required" interface fields should only be set by the ADL / DI mechanism.




```
class Doubler implements Input
{
  private Input input1;
  private Input input2;

  public void displayCharacter(char ch)
  {
    input1.disiplayCharacter(ch);
    input2.disiplayCharacter(ch);
  }
}
```

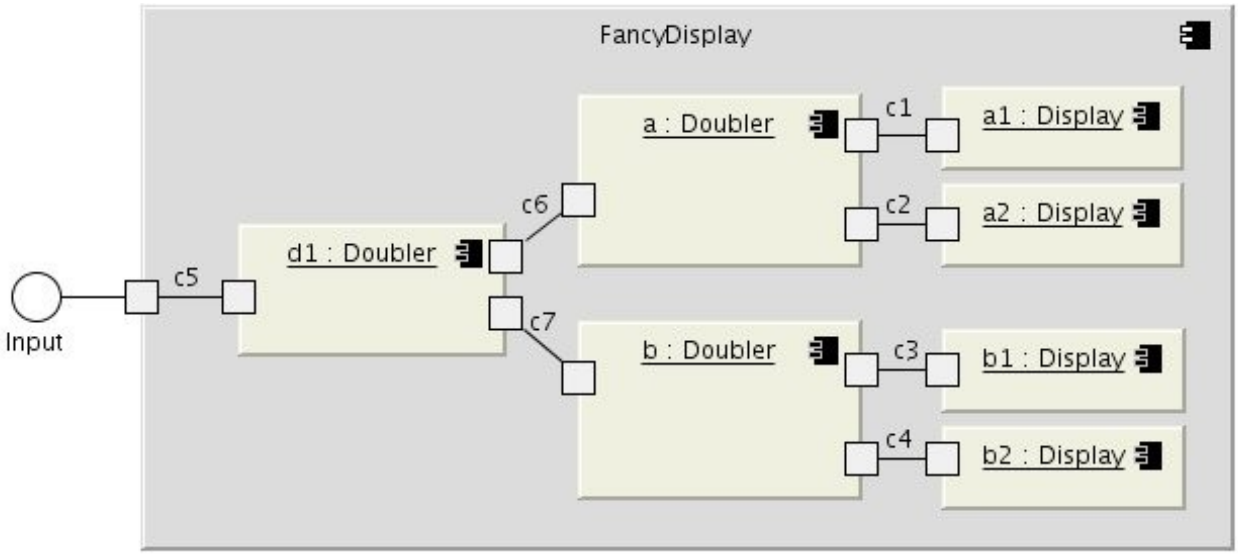
Spring's DI doesn't require that simple components like this be configured. It assumes they conform to the JavaBeans conventions.

Leaf and Composite Components

So Java classes can be used as components. However, if that is all there was to ADL components, then a powerful concept would be missing: the notion of composing increasingly higher level components from lower level ones. This is known as a composite component. A Java class is a leaf component – it cannot be further decomposed into other component instances.

A composite component is a component which is formed by wiring up instances of other components. Technically, it has no behaviour of its own. This wiring up then becomes a new component in its own right, and any internal details and connections can be hidden.

Consider a graphical view of a composite component called FancyDisplay:



The FancyDisplay composite contains four displays, and it uses three instances of the Doubler component to fan out the input to four Display component instances.

Note that FancyDisplay has no Java counterpart but only has an ADL representation. This is because it is just wiring instructions for how to connect together the component instances inside it. In Backbone (the ADL that I developed for my PhD work), the definition would look as follows:

```
composite-component FancyDisplay
{
        ports:
```

```
                inp provides Input;
        parts:
                Doubler d1, a, b;
                Display a1, a2, b1, b2;
        connectors:
                c1joins input1@a to d@a1;
                c2 joins input2@a to d@a2;
                c3 joins input1@b to d@b1;
                c4 joins input2@b to d@b2;
                c5 joins inp to inp@d1;
                c6 joins input1@d1 to inp@a;
                c7 joins input2@d1 to inp@b;
}
```

This facility to compose instances into another component is (predictably enough!) known in the ADL literature as composition. It's a very simple concept, but very profound, because it means that at whatever level you look at a system, you will just see components wired up together. A software system is then fractal-like in nature – the same form at any level of abstraction.

Composition is a powerful way to hide complexity. At the top level, an entire system may be represented by a single component which internally contains N other component instances wired together. If you want more detail, just zoom into each of those component definitions and look at their internal wiring and so on until you hit the leaf components.

Spring is slightly unusual in this respect. A Spring bean must be associated with a class, so technically speaking it does not offer composite components, at least not in the way that most ADLs would understand them. However, it's fairly obvious that you can get a level of composition, because one bean can then become a new component to be wired up inside another bean.

To support composites fully, Spring would have to allow a bean definition to be specified without an associated class definition. The benefits are that it simplifies analysis and system construction. To do this, Spring configuration would need to add explicit connectors and ports. I'll cover the former, but not the latter in the next section.

Explicit Connectors

One feature that distinguishes ADLs from earlier technologies such as module interconnection languages is explicit connectors (see c1 to c7 in the previous diagram). A connector literally “wires” two components together. Making connectors explicit means that much more complex structures can be wired together using the ADL.

Spring has implicit connectors, allowing a bean's property to be set to an instance or reference of another bean instance.

The composite example above is simple to express using connectors, but difficult to do if the connectors are implicit as they are in Spring bean configuration. Spring XML configuration could not wire up the composite component in a single definition. For simple examples, such as this, it is perhaps not a problem. As systems grow in complexity, experience with ADLs has shown that explicit connectors become very important for handling internal component wiring.

Some advanced ADLs even have first-class connectors. This means that connectors act a bit like components in their own right, and can encapsulate a network transport and any error conditions that might arise.

A (Biased and Imperial College-Centric) History of ADLs

ADLs evolved out of attempts to make software architecture explicit as a set of connected components. Conic is an early ADL that used Pascal to implement leaf components, which was created by three professors at Imperial College (including two of my supervisors) in the mid 1980's. Conic used a separate textual configuration language (ADL) to wire up the components. This language is the equivalent of the Spring XML configuration facility. Conic was used to instantiate and wire up components in a distributed system, and supported runtime change through the application of dynamic changes to those wiring instructions.

Intriguingly, the 1985 paper [ref] on Conic first coined the term “configuration programming”. This is a term we are quite (un)comfortable with today, perhaps because of the over-emphasis on XML configuration in Java server-side systems.

The successor to Conic was Darwin which used C++ as for implementing leaf components. Darwin heavily influenced Microsoft's COM, although COM subsequently removed some of the elements that make ADLs so powerful, such as explicit connectors and textual configuration.

Darwin also was the foundation of Koala, an ADL used by Philips for software in some of its consumer electronics such as high end television sets. Koala is an interesting example of the use of an ADL in a very resource limited environment, and like Spring deals with the wiring up of the internals of a non-distributed application.

Darwin, Conic and Koala all feature composite components. COM supports a limited form of composition.

An interesting ADL that doesn't have composite components is Chiron-2, also known as C2. This cryptically named ADL is designed specifically to model GUI architectures. The internal architecture of a C2 component provides a way to ensure reuse in different scenarios. C2 ultimately led to C2SADEL and C2DRADEL, which deal with system evolution and change to a running system.

Unfortunately, most of these ADLs are not usable outside of the academic (or closed commercial) communities. They are quite old now, and had nowhere near the take-up of Spring and other DI approaches.

It may surprise readers to know that the second version of the Unified Modeling Language (UML) contains a perfectly workable ADL somewhere inside the tangled mess of concepts. This subset of UML was heavily influenced by the amazing Objectime (later rebranded as Rational Realtime) and its associated ROOM methodology and ADL. Objectime featured a complete graphical modeling environment also.

The Possible Future of DI?

ADLs didn't catch on in mainstream software engineering. They were perhaps “before their time”, and got swept away in the OO-mania of the period. It also didn't help that they weren't promoted outside of the academic community. I have personally experienced this, as I initially up creating my own ADL independently from any knowledge of the literature, simply because I didn't know other ADLs existed.

However, ADLs were (and still are) a fertile area for research. Basically, much of the work done in the ADL area now takes the form of specification and subsequent analysis. The structure of the system is described using a configuration, and other information is "hung off" the structure. This information might describe the properties of a component, or might be associated with the entire system. e.g. describing and analysing the concurrency properties of an architecture or determining when parts of an architecture are able to be dynamically upgraded in a running system (apologies for the last link – I can't find a freely accessible version of the paper). This information is then analysed for certain properties.

It is from this research that we can learn a lot about how DI might evolve. It's possible that Spring or another DI approach can bring these very useful techniques into mainstream usage.

Please note that I'm not trying to take anything away from Spring or its DI approach. In particular, Spring offers many fascinating facilities (such as aspects) that were never considered in ADLs. My intent though is to look at what other ADL features we might be able to usefully incorporate in future work.

1. Dynamic runtime changes
   When a system is expressed as wired up components, the architecture of the system can be remodelled through changing around the connectors and swapping in and out various components. OSGi and other systems have started to address this in terms of component lifecycle and mechanisms, but DI / ADL approaches are able to deal with this at the architectural level.
2. System evolution and extensibility
   All systems evolve over time, and similarly this can be represented by switching components and changing wiring. However, advanced ADLs are able to capture this information by overlaying the changes on the original design (using architectural deltas). This allows the new system to be expressed as changes to the old system, allowing someone other than the original creator to evolve or modify it without destroying the original version in the process. Some of my work deals with this area.
3. Analysis of Behavior and Protocols
   Since components communicate via interfaces, it is possible to model the protocol of a component (and interfaces) using an extended sequence diagram or using a simple textual language. This is basically "behavior driven design" for components. This information allows components to be checked when they are placed in a different configuration. i.e. Will a component function as intended?
4. Concurrency analysis
   There has been much work on checking the operation of components in a concurrent environment. In fact, entire books have been written about it. Concurrency bugs in large systems are notoriously hard to detect and remove, but the techniques in this area offer a relatively lightweight way to find these through analysis.
5. Distributed systems
   The earliest ADLs were designed to instantiate the architecture of complex distributed systems and monitor the nodes making up the distributed network. Current DI approaches

only wire up the internals of a single application.

6. New Domains
   DI is mainly used to wire together server side components. C2 showed that even client software could benefit, and encoded common GUI patterns within its structure. Perhaps this has already happened in Spring RichClient, and I've just overlooked it. There are a wealth of other domains to mine.
7. An enhanced component model
   Adding full composite components and explicit connectors will add to the expressiveness of current DI approaches. Explicit ports would also be useful, as they complete the model for composite components, but are outside of the scope of this short article.
8. Combinining components and modules
   Although ADLs to some degree were seen as replacing modules, the need for modules didn't go away. Modules are used for packaging and deployment, and for distinguishing between the private, implementation part of a system and its public interface. Spring has gone some way to addressing this already with the OSGi integration. It will be interesting to see where this work goes.

Summary

Dependency injection is here to stay. It is related to component techniques and architectural approaches that have proven themselves in the commercial and academic arenas for almost twenty years now. There are other aspects of ADLs that may be useful in the future to consider for DI, particularly the approach towards analysis, and a more complete component model.

Sidenote

I started writing an online article about my PhD work, which is on system extensibility and evolution using ADLs. However, I decided I really needed to write this article first and get some of the background points out there first. I wanted to do this because there appears to be little interest in DI in academia (been there, done that, academics don't usually care about take-up) and no visibility of ADLs in commercial development (looks too complicated, not production ready, inaccessible).

It struck me as I was writing this article that there is a huge divide between software engineering practice and research. Whilst popular practical techniques will eventually form the basis for research, and some research will eventually filter down into practice, there doesn't seem to be a large interplay between the two communities, which strikes me as both surprising and disturbing. I'd be very interested to know your thoughts on this divide and whether it affects you in your work.