Wavefront Extended Abstract

Code to Last 100 Years?

Infusion IoC, a JavaScript library and mentality for delivering accessible and maintainable systems

Antranig Basman
Fluid Project, OCAD University, Toronto, Canada
antranig.basman@colorado.edu

Clayton Lewis Colin Clark
University of Colorado, Boulder/Fluid Project
clayton.lewis@colorado.edu/cclark@ocad.ca

Abstract

As soon as a developer using the techniques of today opens his mouth to utter some code, he is condemned to a depressing and familiar train of events. The best he can hope for is for his code not to survive — more often than not, through blind necessity, this blissful release is not achieved. Code and designs become "old" within very few years — often even at the point of origination. Old code is brittle and hard to refactor, hard to press to new purposes, and hard to understand. Here we present a system aimed at creating a model for *scalable development*, addressing this and several other critical problems in software construction. Such an aim is far from new, and has resembled the aims of each generation of software methodologists over the last 50 years. It deserves comment why these aims have so signally failed to be achieved, and we will present arguments as to why the combintation of techniques explained here could expect to lead to novel results.

Some categories of failures to be addressed: software products of today are notoriously unadaptable — an application which meets need A is unlikely to be able to be extended to meet apparently very similar need A' without something resembling "software engineering". It is hard to reason from "effects to causes" — on seeing an effect in an interface which a user considers either desirable or undesirable, they are unlikely to be able to mount a successful interaction with the system aimed at either preserving it or removing it. Successive revisions of software present users with a "take it or leave it" proposition. Finally, software fails to be easily adaptible to meet the needs of users with differing requirements — "accessibility" work is performed as an afterthought, if at all, and often these needs are met by developing a largely unrelated version of the application.

We will present a model for software construction, together with a base library implemented in the JavaScript language. This features a notion of *context* as the basis for adaptibility, a scope implemented neither lexically nor dynamically, but as a result of the topology of a spatialised data structure, a *component tree* expressing the computation to be performed. We will also work with a model of *transparent state* in which the total modifiable state within a tree is held at publically visible addresses, indexed by path strings. This model for state is isomorphic to that modelled by JSON, a well-known state model derived from, but not limited to, the JavaScript language. The instantiation engine is an *Inversion of Control* system extended from the model of similar system such as the Spring Framework or Pico first developed in the Java language. We relate such systems to goal-directed resolution systems such as PROLOG, and their recovery of beneficial properties of code such as *homoiconicity* which have not been seen in a strong or widespread form since the days of LISP. We will exhibit some cases to show how the framework enables, through a simple declarative syntax, types of adaptation and composition that are hard or impossible using traditional models of polymorphism. We will conclude with some remarks on the applicability of the system to the parallelisation of irregular algorithms, and relationship to upcoming developments in the ECMAScript 6 language specification.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords JavaScript, Inversion of Control, Transparent State programming, Accessibility, JSON,

1. The Development and Need for Inversion of Control systems

1.1 The Crucial Nature of Dependency

John Lakos[**?**] produced one of the clearest discussions of the impact of dependency on a design. In his conception, a piece of code A has "knowledge of" or "dependency on" another, B, if names in B appear in A. To be precise when using such words in his sense, we will qualify them by referring to "L-dependency" or "L-knowledge". In the C++ language in which Lakos was working, there are various gradations of knowledge, for example, whether the knowledge about B was sufficient to affect the memory layout of objects allocated in A, or merely required the compiler to have visibility of B names when compiling A code. Although the details differ, the core of this formulation is invariant across essentially all programming languages.

Lakos argued that code in a "dependency-correct" system should form an *acyclic graph*, when expressed in terms of the logical units into which it was divided and the L-dependencies among them. In the C++ language, these logical units were often classes, although he noted that this kind of boundary could be drawn at any level in a system.

Lakos observed that there were many more significant consequences of constructing bodies of code with inappropriately arranged L-dependency than simple increase in build times. Highly interdependent code was harder to understand, harder to test and maintain, and most importantly to our domain of end-users, tended

to be extremely brittle over time. Such code imposes unexpectedly huge development costs to respond to seemingly innocuous feature requests.

As it turns out, the problem raised by Lakos' recommendations on the organisation of dependencies cannot be adequately resolved in the C++ language at all. Consider a hypothetical DAG of dependency-correct code, organised into units, say, of classes. Take two of these elements, A and B - in terms of C++, knowledge of class A about class B, would translate into a requirement for objects of class A to bear responsibility for construction of objects of class B, and not vice versa. This knowledge may actually be pushed into a common ancestor, C - but wherever it resides, this constructional knowledge cumulates towards the root of the tree, creating a *fragile base* to the overall design.

Common solutions to these issues in non-dynamic languages invariably involve one or more constructional "design patterns", more commonly factories. These impose two kinds of penalties - firstly, a tyranny of "common constructional capabilities" imposed by the limitations of polymorphism in static languages - in order to represent an axis of supportable variability, the requirements to be met by a family of products from a factory need to be expressed in terms of a common signature, becoming the signature of an interface or base class managing the products. Secondly, whilst *some* type information may be erased at this polymorphic boundary, this also naturally cumulates upwards towards the base of the DAG of knowledge, this type information also leading to a source of scaling failure in dependency and type-correct designs.

1.2 Inversion of Control Systems

The Java language is not particularly dynamic, but enjoys enough of this quality through its reflection system and the possibility for bytecode manipulation that some workable solutions to the fragile base problem emerged. Martin Fowler outlines some of the variants of IoC framework in [**?**]; popular frameworks in Java include Pico, Avalon, and currently most popularly the Spring framework[**?**].

The conception behind these system relies intrinsically on dynamic properties of the target language. If an object of class A needs an object of another class B at construction time, rather than A's code calling a constructor for B, A's need for a B is registered in some kind of declarative format with the IoC system, and then the IoC system **injects** an instance of B into the object that needs it. The "inversion" is that "asking for an object" is replaced by "being given an object". In fact, rather than "constructing itself" as is the case in static languages, the entire tree containing A, B and all neighbouring dependencies is constructed by the framework, informing the target code of lifecycle points in a model similar to that of event-driven frameworks.

Users of these frameworks get increased agility in the face of end-user requests and variability in environment. That is, important environmental decisions (in the concrete terms of workaday developers, issues such as transaction management, database dialect, message resolution etc.) are taken out of the code and replaced by declarative configuration.

1.3 Limitations and Extensions

A significant lack in existing IoC systems is a suitably flexible concept of context. To a Java IoC system, the context is a **container** — a configuration file is entered into the system as its "global rulebase" and if users or developers require changes in resolution based on recognition of a new context or requirement, they need to change the file. Even organising such files hierarchically does not permit decisions to be made based on dynamic considerations. But we can extend the notion of IoC to allow contexts as well as tasks to shape what a system will do.

The Fluid IoC system supervises the matching of names of functions to implementations. What we speak of as a **function name** is more generalised than the traditional notion of a "function" in that it does not necessarily correspond to a function as implemented directly in the programming language — although all names of such functions if registered globally could serve as "function names" if required. Instead a "function name" corresponds to the notion of a "task to be performed" — in the world of a user. (In the tower of abstractions, users operate at different levels, where the definition of a task at the level of one user, say and end user, decomposes it into subtasks that make sense only to a user at another level, say an application designer.)

An implementation provider, and furthermore, unrelated third parties, can provide a set of directives to the IoC system, which specify under which conditions a given implementation is an appropriate one to deliver to an end user who asks for a given function. These directives are named **demands blocks**, matching conditions which are represented by supplying one or more **context names**. These names are also simple strings, like function names.

The power of the system to proceed in a contextually aware way is significantly enhanced by allowing the names of *products* of the system to serve as names of *contexts* guiding the construction of future products — some names may serve as both function names and context names. The name of a user interface widget, for example, may be used sometimes to specify needed functionality, and sometimes to specify a context in which a subsidiary widget might be embedded.

1.4 Link to Goal-Directed Programming

One way of understanding the cascade of instantiations performed by an IoC system in pursuit of constructing a particular "object", is as related to the "resolution" process performed by knowledge-oriented systems such as Prolog.

An important movement in codifying human knowledge of the world was represented by approaches starting with the Prolog language created by Alain Colmerauer and his group in the early 1970s. This casts knowledge in the from of **relations**, connecting one term with another. The input from the user proceeds "forwards" in their world, expressing the dependence of one proposition (or alternatively seen, "goal") on another. E.g. "in order to know whether I will go out today, first I must know whether it is raining or not". Each "rule" of this kind is entered into a database of such rules progressively, building up an unbounded network linking these propositions. A "run" of the system takes the form of requesting the status of a particular proposition - execution then cascades "backwards" (in the view of the developer) through the set of dependent rules until an answer can be determined.

Prolog enjoyed a limited success in building bridges between the semantic worlds of end-users (or "experts") and implementors, but was constrained in a number of important respects. Firstly, its reliance on logical terms for its implementation domain seriously constrained its usability for expression over the full range of tasks humans might wish to address. Whilst conventional data structures can be mapped onto Prolog constructs, this mapping is not easily recognisable by mainstream programmers. Since the official end result of a "pure" Prolog program simply consisted of "true" or "false", interaction with an external, stateful world was typically hacked on as an impure addition to the base system by means of predicates with side effects.

The second area of limitation is also similar to the one we just outlined for IoC systems - Prolog supplied no natural concept of a "context" or "scope" for a body of knowledge - this was all assumed somehow to be "global" and refer to a "general condition of the world". As a simple example, Prolog provided no straightforward means for dealing with situations which changed over time.

However, as humans, we all routinely navigate multiple realms of knowledge where the same names may be given significantly different or even contradictory referents, perhaps with overlapping islands of consistency which can be used to convey results from one realm to another. Our work in this community in fact consists almost entirely of such a work of translation. An corollary of this lack of context-awareness is Prolog's lack of standardised support for "programming in the large". Whilst various module systems were produced, these are not consistently implemented, and large Prolog programs rapidly become unmanageable and brittle as a result.

1.5 The Crucial Important of Homoiconicity

The "curse" of code manifests itself most concretely in its traditionally concreted-in position in a processing pipeline. By the "central dogma of programming languages", code progresses unidirectionally from its form in a text file produced by someone resembling a "developer", through to lexing and parsing stages, to representation of an AST which through various further transformations and optimisations results in object code which is linked to become executable. Although many erosions and shortcuts exist in various environments, this is the basic workflow in which most software practitioners live their everyday lives. All of these stages are completely antithetical to any conception that someone in the real world who wants some work done has of their task. Some environments "cut" this workflow by either producing interfaces for "end users" which synthesise source code, or producing libraries which allow some limited domain of problem be handled by a de facto "domain specific language" represented in data structures held by the program. In neither case do these results produced by end users have any helpful or reversible relationship with the method of choice that would be adopted by a software professional addressing the same task.

In our aim of "building bridges" between the worlds of software professionals and people who want work done, we argue it is essential that some form of bidirectional transfer of artefacts is possible, from end to end of the spectrum between those of the highest level of technical sophistication implementing libraries, and those cast as "end users" only working with the finished product. This set of transfers should not be "mutually blind" but allow some form of harmonised understanding of the transferred abstraction — the system should exhibit a "homogeneous tower of abstractions", stretching from the low levels out into the world of users.

A crucial element of a software system that can be worked with in this way is a "self-understanding" of the syntactic structure of the language, that allows the process of "software operating on software" on behalf of a user to proceed as part of such a homogeneous system. This property has been given the name of *homoiconicity* — whilst many languages lay some form of claim to this property, few approach even closely the level enjoyed by one of the earliest of computer languages, LISP. In LISP, a "program" consists of an "S-expression" which may be viewed equally as an executable element of the language, or else as a data structure known as a *list*. In LISP, programs known as *macros* may operate on lists, interpreted as programs, and transform them into new programs. Many subsystems, such as CLOS, Flavors, LOOPS, etc. were built upon the base of LISP, but the basic homoiconic structure was never built on or expanded. LISP contains the foundation of the "homogeneous tower" we mention, but they do not stretch out very far, and are quite narrow in that the primitives of the language are somewhat impoverished, consisting of just one structure-forming primitive, the list, which impedes interpretability and readability of the language.

Our system builds upon this conceptual heritage by defining several dialects of the state-oriented subset of the ubiquitous JavaScript language, JSON, which may be viewed as direct representations of ASTs of a hypothetical language. The power of LISP macros, then, to reflect on and transform program material, is in the hands of standard Javascript programs operating on these structures. At the end of the paper, we will appeal to the design of a future, strongly homoiconic language in which these representations will directly be interpreted as syntax trees, and may operate on themselves without the visible intercession of a JavaScript-like language.

2. Implementation territory

2.1 Demands blocks

The core building blocks of the Infusion IoC system are JSON structures known as "demands blocks". These direct the resolution of a particular function, in a particular context. These functions, which may or may not correspond to the names of concretely available functions at the language level, are issued from a particular context in the tree of components, which itself is also considered to be a freely addressible complex JSON structure. Whilst the component tree may consist largely of freeform JSON material, it is structured in units which are recognised as "components" per se, which may hold material interpreted in other JSON dialects. Precise conditions on interpretation of the component tree material will be presented later.

As well as the ability to redirect the dispatch of the required function name held in the demands block, the arguments to the function call may also be freely interspersed, replaced, or merged with material drawn from elsewhere in the tree. This resolution may occur both at the initial construction time of the tree, guiding functions interpreted as *component creator functions*, or at subsequent times during its lifetime, interpreted as *invokers* or *events*.

```
fluid.demands("fluid.uploader.local", "fluid.uploader.html5Strategy", {
    funcName: "fluid.uploader.html5Strategy.local",
    args: [
        "{multiFileUploader}.queue",
        "{html5Strategy}.options.legacyBrowserFileLimit",
        "{options}"
    ]
});
```

Figure 1. Sample of a demands block

The first argument to `fluid.demands` holds the *demanded function name* - the name issued from the tree which this rule is intended to match. The second argument holds one or more *context names* which are used to scope the matching of this rule — in order for the rule to match, components holding these names need to be "in scope" at a suitable location in the tree of components from which the original function name is issued. The third argument is a JSON structure which expresses the disposition of the function call in the case the rule matches. In this case, both the function name and argument list are redispatched — three arguments are synthesised from material available in the environment of the call. In this material, contextually resolved names (in the same namespace as the context name of the 2nd argument) are set off by braces {}, followed by an optional path selector resolving some subobject of the matched context object. Some context names, such as {`options`} have special meanings referring to either the original argument list or declarative material at the call site serving the same purpose.

References

[1] Lakos, J.: Large-Scale C++ Software Design, 1996, Addison-Wesley Professional

[2] Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern, `http://martinfowler.com/articles/injection.html`

[3] Douglas Crockford — The JSON Saga: http://developer.yahoo.com/yui/theater/video.php?v=crockford-json