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

Using the techniques of today, code and designs become "old" very quickly, often unmaintainable from the point of inception. 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 combination 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". Successive revisions of software present users with a"take it or leave it" proposition — if the software doesn't happen to meet a user's needs or preferences, there's no way to change it without writing more code, which is out of reach for most users. Indeed, software regularly fails to be easily adaptable to meet the needs of users with differing needs, such as in the case of accessibility. These "precarious values" — accessibility and usability with different devices, languages, and personal needs — are typically left until the end or ignored, and represent a significant expense in traditional approaches to software development. Often these needs are met by developing a largely unrelated version of the application, requiring maintenance of additional, separate code bases.

To address these problems, we will present a model for software construction, together with a base library, Fluid Infusion, implemented in the JavaScript language. This model features a notion of *context* as the basis for adaptibility, resolved in a scope implemented neither lexically nor dynamically, but as a result of the topology of a 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*[2] 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

The system described here, Fluid Infusion, is an "Inversion of Control" system implemented in the JavaScript language. It constructs applications from trees of components expressed declaratively in JSON notation. Whilst the primary use of the system is in assembling user interface markup and operating logic for HTML web applications, these ideas and Infusion's implementation of them can be adapted to other domains, illuminating broad issues of software construction. We will begin by examining the history and motivation of similar systems, the relationship of our IoC system to other models of software construction, and then finish by describing some current applications, the current state of the Infusion framework, and planned future work.

1.1 The Crucial Nature of Dependency

John Lakos[1] 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 a *directed acyclic graph* (DAG), 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 significant consequences of constructing bodies of code with inappropriately arranged L-dependency, following on from his initial effort to control escalating build times in complex systems. 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 fully resolved in C++, or other static languages, 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 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 attempted solutions to these issues in non-dynamic languages involve constructional "design patterns", usually factories. These impose two kinds of penalties. Firstly, the family of products from the factory need to have a common signature, a serious restriction. Secondly, whilst *some* type information may be erased at this polymorphic boundary, remanent type information still naturally cumulates upwards in the DAG of knowledge in a way that prevents scaling.

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, generically described as "Inversion of Control". Martin Fowler outlines some of the variants of IoC framework in [3]; 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. The IoC system then **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. The IoC framework, in this model, takes the place of the brittle constructional code otherwise placed in class C.

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 a global specification 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. All names of such functions could, however, if registered globally 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. There are generally different classes of "users", operating at different levels in a the tower of abstractions, where the definition of a task at the level of one user, say an end user, decomposes it into subtasks that make sense only to a user at another level, say an application designer.

An implementation provider — and even 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. 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. Prolog 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. 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.

Recursive resolution of dependent components by an IoC system can be seen as a model of a similar process as the cascade of Prolog relation resolution. Important differences are that whilst this IoC system currently operates no form of "backtracking", we add a concept of **context** to the resolution system. Absence of contextual awareness was historically a weakness of Prolog, which, for example, provided no straightforward means for dealing with situations which changed over time.

1.5 Link to Aspect-Oriented Programming

A popular approach for dealing what it terms "cross-cutting aspects of a design" which has grown up alongside and in some cases intertwined with the use of IoC is known as "Aspect-oriented program-

ming". In this model, the implementation domain of a codebase is stratified, forming a higher "meta-level" of design comprising units of code (in a related, but usually distinct syntax) which consists of directives which *advise* the operation of the remaining base level of code which can usually enjoy some kind of simplified implementation.

AOP systems are often extremely powerful, and have the ability to issue *advice* which modifies the execution of the base code at the level of individual method calls or property access - either modifying this dispatch or replacing it entirely. However, this power of oversight, whilst broad, can often be "blind" or at least shortsighted — the specification of a "cutpoint", the environment in which an advice matches, is made in quite low-level terms, and with the data-hiding mentality which goes together with object-orientation, usually have quite limited insight into the contextual situation which has been matched. As a result of the incredibly broad power of cutpoints to match, but limited power to act, AOP designs can become very hard to understand without custom tools and the temptation to use advices extensively is strong.

The "redispatch" formed by the matching of Infusion IoC demands blocks has a similar kind of power, but is at the same time limited in its scope for matching, as it is broadened in its ability to interpret context. A demands block can only act at points in a design where the IoC system is already instantiating a subcomponent in the tree, or else where the user has explicitly requested its operation by use of an *invoker* or *boiled event*. However, when it does act, the dispatch modification may make use of the same contextual resolution system which guided its own matching, to stably discover relevant pieces of state over the entire component tree in scope, rather than just those located close to the advice site as in traditional AOP. This tradeoff of increased formality of matching against increased contextual understanding should produce designs which are much easier to understand as a whole, although we still anticipate a very important role for assistive tools. However, since these tools only require parsing of a simple dialect of JSON rather than the free-form mixture of advice syntax with the base language as seen in AOP, we anticipate them being much easier to write.

1.6 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*.

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

```
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

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.

2.2 Case study — Progressively enhanced Uploader component

Whilst the IoC system to some extent allows us to get beyond traditional conceptions of "component–oriented" or "object–oriented" applications, it is an important stepping–stone in providing value to users and developers of the system alike to develop packages of functionality that fill well-defined, though flexible functions. An application that we have worked on recently and present here is that of a "Uploader widget", that whilst presenting an straightforward and stable entry point to users as a simple function call, intelligently adapts to the combination of the capabilities of the browser it is instantiated in, as well as the users expressed preferences. At the same time, this implementation can be extended to deal with unanticipated technologies and environments, without modification of either the implementation or user code. In its simplest possible form, the use of the widget can proceed as follows (assuming inclusion of appropriate Fluid and jQuery JavaScript files):

```
<html>
 ....
    <form method="post" enctype="multipart/form-data" class="myUploader">
        <input name="fileData" type="file" />
        <input type="submit" value="Save"/>
    </form>
    <script type="text/javascript">
        fluid.uploader(".myUploader");
    </script>
</html>
```

Figure 2. Sample instantiation of an uploader component

In this simple form, a call to a concretely named JavaScript function is targetted at a block of markup holding a standard (HTML4-style) file upload form. Since no further configuration is supplied, the IoC machinery behind the function call will detect whether a Flash version 9, Flash version 10, HTML5 (Firefox 3.6-style) or Binary XHR-compliant HTML5 style (Firefox 4 or Chrome) uploader is the most appropriate form, inject appropriate markup and construct a suitable implementation. Should none of these choices be workable, or further, should JavaScript not even be enabled in the browser, the markup will be left as it is, and still fulfil the basic function of allowing selection and upload of a file.

The variant "enriched" implementations all present a common interface, allowing selection of multiple files, feedback of individual and overall progress as well as pausing or cancellation of operations — and naturally this implementation code, as well as several other sections of the implementation are reused across the variants as "invariant sections" of the component tree. Even this relatively compact problem would have been hard to address through standard "object–oriented" techniques — assembling a class hierarchy mapping this area would have run into two principal hazards:

Firstly, the pattern of code reuse, with multiple, mutually–overlapping pieces of implementation shared between the configurations would have been hard to map onto one (or more than one) traditional inheritance hierarchy. Even were this done, it would be hard to be assured that in the face of future variation, the mapping would remain stable — in our IoC implementation, since essentially all implementation which is in code is packaged in a form equivalent to that of "free", context-less functions, it is easy to be assured that it can be recomposed by means of demands blocks into a new and suitable arrangement where implementation can be common.

Secondly, the simple point of entry in terms of a stable, context-less function `fluid.uploader` would have been harder to achieve. Given the delivered implementation is polymorphically variable, this would either have required concrete type-names to be mentioned in a "constructor", or the use of some suitable "factory method" on an already existing source of implementations... whose construction would represent the same problem, pushed back one level.

Say that, for example, despite the now obsolescence of Google's "Gears" technology for native browser functions, a user wanted to extend this Uploader implementation set to support it. This could be done without modification of either the user entry point `fluid.uploader` or the implementation files. In fact, all the demands blocks and implementation functions required for, say Gears, or some unanticipated future technology could be scoped to a JavaScript file which is not even delivered to users with user agents not supporting this technology. This kind of "file inclusion-based polymorphism" is hard to package with object–oriented techniques where constructors delivering implementations typically need to appear within the delivering code.

Figure 3 shows schematically the structure of some of the demands blocks implementing the Uploader widget. Each arrow linked with a circle along its length represents a *contextual resolution* — a choice made by the instantiation system based on the context provided by constructions which have already concluded. At the top of the diagram are raw **context tags** decoded by a direct inspection of the capabilities of the user agent — `fluid.browser.supportsBinaryXHR` etc.. Below this level in the tree is the point at which user configured material may be used to guide resolution, for a particular instantiation of the uploader, onto a particular strategy to be used — in the absence of this, the default demands structure will proceed by a default algorithm onto the uploader strategy tags, `fluid.uploader.html5` etc. Below this level, the resolution continues to cascade onto particular elements of the uploader implementation, guided by the strategy tags. At each level of demands resolution, there is scope for further demands blocks contributed at the user's request, or an integrator or other party, to intercede, or in AOP terminology to "advise" the construction of subcomponents, by fetching data or implementation sourced from other parts of the uploader's tree — or even, from other parts of a wider component tree. Whilst the uploader is packaged in such a way that it is usable by standard JavaScript citizens as a simple function tree, the full power of the IoC system together with the uploader is only realised when the entire implementation of an application is delivered as a single, giant interconnected series of demands blocks — with demands resolution given the power to roam freely and fetch contextualised data to be delivered anywhere within or among elements which would formerly have been seen to be opaque, monolithic "components".

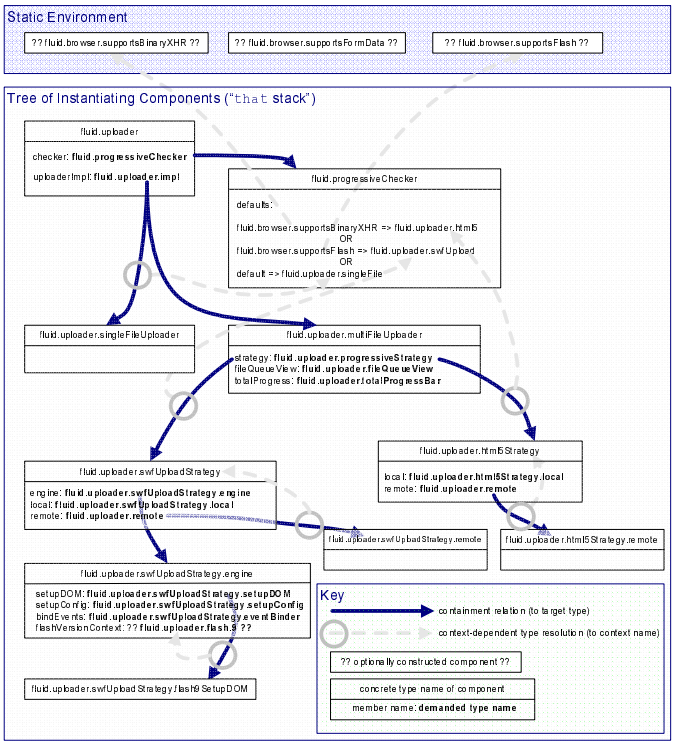

Figure 3. Illustration of component tree instantiation for Uploader widget

Whilst Figure 3 resembles a standard UML diagram in some respects, the meaning is somewhat different. This diagram shows a schematic for data structures which *might* exist in certain histories of the system, rather than those which will or do exist, statically. It is this contextual awareness at each point of the system that allows it to be easily extended for new cases, with instantiation guided along new paths, made visible by new demands blocks being brought into scope.

2.3 Wider case study - The CollectionSpace Collections Management System

The CollectionSpace project[5], led by the Museum of the Moving Image in New York, is producing collections management software for the use of museum curators and other staff. The project is using the current version of the Infusion IoC system, as described here. This project is proving an excellent grounds for exploring the benefits in adaptible and declarative software that the IoC approach can offer. The user interface for this application consists of few but very detailed pages, containing many hundreds of widgets, reflecting the level of detail of the specialised knowledge operated in the domain of exhibit curation. As suggested in the previous section, the entire UI for this application is indeed implemented as a giant, single-rooted tree of components governed by the underlying graph of demands blocks, instantiating components such as the Uploader and numerous others in an IoC-driven way. One immediate benefit of this approach for users is the easy adaptibility of the interface in a schema-driven way. Rather than rely on development support to orchestrate changes required by local institutions which may have very widely differing requirements, these can instead be enacted by editing of simply-structured JSON files or in many cases be inferred automatically from a description of the application's schema.

Similarly, with component markup not locked up in implementation files but "out in the open" in unpolluted and standard HTML files, reskinning of the application similarly can be performed without development support, using standard HTML editing tools. These kinds of reskinning clearly include, but go beyond that possible through simple CSS effects, into widespread reorganisation of the layout and content of the markup operated either by individual components or entire pages.

2.4 Status of the implementation and framework

The Fluid group are currently working towards the 1.4 release of the Infusion system, which is targetted for the beginning of May. This will be the first public release in which the described implementation of the IoC system (as well as the Uploader widget and other components not described here) will be available. Readers are invited to come along and inspect our progress, and even join in, at our github repository held at `https://github.com/fluid-project/infusion`. Overall documentation for the Infusion system, including the IoC implementation, is held at `http://wiki.fluidproject.org/display/fluid/Infusion+Documentation`. Future versions of Infusion are roadmapped at `http://wiki.fluidproject.org/display/fluid/Fluid+Community+Roadmaps` — we will continue to stabilise and expand the capabilities of the IoC system as well as evolving previously implemented components to defer to it more for implementation. A crucially important, but still very early–stage work package involves our server-side implementation, Fluid *Kettle*, an IoC-driven JavaScript serverside implementation based on the rapidly developing `Node.js` framework based on an asynchronous I/O model. A back-end based on Apache's CouchDB persistence technology using JavaScript as a query language will enable a homogeneous development model operating JavaScript at all tiers of the web application, which is hoped to bring developments of lowered barrier to entry by new developers as well as increased mobility of code and implementation algorithms between the layers.

2.5 Conclusion

Whilst the code presented here may not individually survive for 100 years, we have made a case arguing that the longevity of application code, its useful working life where it can continue to be adapted to new tasks without degrading its infrastructure, is greatly increased by reducing as much of its volume as possible to a declarative form — a promising model for such a form are the JSON blocks we have described here, forming the demands and defaults blocks interpreted by the IoC-driven component system. As platforms and technologies change, new demands blocks can weave together with the old to meet new needs, without fragility in existing implementations. Should JavaScript and the web themselves cease to become current, this declarative form is easier to mechanically transform (following the mentality of LISP "macros") into forthcoming idioms, than implementations specified in imperative, sequential code. Such code that *is* written is packaged in global functions which are more or less "free", maximising the chance that it can be reused in fresh contexts without the worry of assumptions embodied in hazardous shared state such as that found in base classes or object instances. Finally, where expectations and contracts do change over time, old implementations may be adapted to new clients, and vice versa, by the interposition of suitable demands blocks, providing the appearance of new contracts for old.

References

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

[2] McIlroy, d: Macro Instruction Extensions of Compiler Languages, 1960, Communications of the ACM, Volume 3 Issue 4

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

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

[5] The CollectionSpace Project: http://www.collectionspace.org/