A New Open/Closed Principle

Additive Supports for Networks of Creators

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

We introduce a new *open/closed principle* which establishes desirable design characteristics of languages and configuration systems supporting networks of creators. We survey the growth in power in authorial systems, situating them in a progression starting with traditional object-orientation, continuing with aspect-oriented, context-oriented, and dependency injection systems, and concluding with the most recent generation of "freely dimensioned" systems such as Korz and Fluid Infusion. We rework an example originally developed for Korz into Infusion's configuration system, and discuss how multiple authors can contribute fresh implementation dimensions into the same artefact, and how priority amongst their contributions can be resolved. We exhibit a working system that allows adaptations to be dynamically contributed into a video processing pipeline, and conclude by discussing how systems may be designed to facilitate the provision of "avatars", replicating the function of a (spatial) segment of an implementation for a bounded period of time.

Keywords context awareness, declarative configuration

1. Networks of Creators

Successive developments in the structure of programming idioms appear to be aimed at granting increasingly powerful capabilities to those seeking to reuse and repurpose the artefacts of others. The original open/closed principle(Meyer 1988)[1] , codifying what is now accepted as one of the core principles of object-orientation. Meyer's principle allows for what could be described as a form of "first-order reuse". This provides only for reuse of single implementation elements at a time (classes/objects), but does little to facilitate reuse across a design or with larger aggregations — in fact, it could be argued that object-orientation actively impedes this form of wider reuse, as we discuss below.

[1] To paraphrase, "A module should be available for extension (*open*) but available for use (*closed*)" — where Meyer connoted availability for use with the fact that a module's content should not be modifiable, promoting uses such as caching, verification, etc.

To situate this discussion we will examine scenarios of repeated reuse within the context of a *network of creators*. Such a network is a directed graph with creators or sites of reuse at the nodes. Two creators A, B are connected by an arc if a tool or other community channel of reuse allows B to base his/her artefact on A's artefact. One example of such a connection is whereby A writes source text which is processed by a compiler lying along the arc, resulting in an executable used by B. Another is if A writes a base class which is imported by B in order to produce a derived class by the addition of source text.

We will enumerate increasingly sophisticated reuse scenarios in a numbered sequence, starting with level 1 reuse representing that enabled by Meyer's principle. As wider networks of creators attempt to collaborate and share authorial access to the same, increasingly rich, artefacts, the reusability requirements on the language and authoring infrastructure increase in level.

In this presentation of reuse levels, we'll avoid the use of idiom-specific terminology for naming implementation units such as "object", "class", "module", "type" and so on, to avoid biasing the discussion.

2. Meyerian Reuse - Level 1

In Meyer's presentation, there are implicitly creators, A, B, E. Note that Meyer does not explicitly assemble this named network of creators — we reconstruct it from the needs that object-orientation are exhibited to meet in his presentation[2]. A has created an implementation unit named α. B wishes to refine α to α', and share this with E as a substitutable replacement for α. B is assisted in doing this by a language feature of an object-oriented language granting the ability to explicitly or implicitly create an artefact taking the form of a

[2] We use Meyer as a standin for the much wider community sharing the same reuse model, such as the Smalltalk/Self communities tracing lineage to Kay and the mainstream Java/C++/C# communities, etc.

base class or *interface* $\aleph$ which expresses some of content of the contract A and B advertise to E. When E's use of α or α' is confined to the scope of $\aleph$, B's modified code providing α' can be swapped for A's providing α without E needing to know about the change. Note that this swap can only be made in terms if E is using a α that already exists; their code can equally well use an α'. But if E's code creates an α, it can't instead create an α' without being modified. Typical solutions in object-oriented frameworks to this problem of constructional dependency resolve around the use of one or more varieties of "factory pattern" which we will return to when we start to treat more profound incarnations of reuse problems in the the following section 3.1.

3. A Basic Reuse Scenario - Level 2

In this section, we'll explore the most basic elaboration of the Meyerian (level 1) reuse scenario that exposes the limitations of object-orientation and other contemporary idioms. Creator A has created an implementation unit named α which contains a subunit named β. Creator B has refined β to β', and wants to create α' which is α with its β replaced by β'. B would like to share α' with creator E as a substitute for α.

3.1 Containment through aggregation

There are several possible embodiments of the "containment" relation here, even within the same idiom, which lead to somewhat different fates for B's plan. Firstly, containment may be aggregation — α is a class which has a member b of type β. In unadorned object-orientation, we are already out of luck. We have no way to modify the place in α where b is declared, without simply rewriting the code — "open to extension but closed for modification" has failed. Some traditional responses to this problem require one or more varieties of "factory pattern"(Gamma 1994) — the creation of an "intermediary artefact" we might name $\beth$ whose purpose is to abstract over the construction point of β. This is obviously unsatisfactory [3] since we have a whole extra class of entity to design in the system, in practice with its own type hierarchy to be maintained in parallel with the base artefacts — as well as a wholly unmanageable "regress" problem of how the same problem with respect to the factories is to be resolved. Other responses to this problem require a fresh language feature, orthogonal to the classically object-oriented ones, allowing the expression of "parameterized" or "generic" types — we'll return to this possibility in section 3.2.

3.2 Containment through inheritance

Another possibility relates to strategies for "self-aggregation" which in object-orientation are embodied as the inheritance relation where we might say that α "IS-A" β through including its entire definition into its own. Our reuse problem is particularly recalcitrant via this relation in O-O and essentially forces the requirement for parameterised types to be added to the system.

[3] "*entitia non sunt multiplicanda praeter necessitatem*" as Occam observed

3.2.1 Reuse through parameterised types

In C++, creator A, perhaps trying to address the reuse situation of section 3.2 would have had to have *already written*

```
template <class beth> class alpha: public beth {}
```

so that they themselves could then write

```
alpha<beta> myBeta;
```

and that creator B could write

```
alpha<beta1> myBeta1;
```

Creating a template like this requires foresight from A: they need to anticipate that someone in their community may wish to modify β. It also adds complexity, as type signatures become longer and more involved. The name of an α simply cannot be mentioned without also bringing the requirement to mention the particular $\beth$ it involves.

Parameterised types are a sufficiently powerful reuse mechanism that they also resolve the aggregation variant of this problem in 3.1.

3.3 Containment through private use

Another possibility for the meaning of "containment" is that the point within α where β is used lies simply within implementation code — for example, the body of a method, and the β instance does not appear within the class definition. This situation is yet worse than the one before, since we not only have to refactor α but also rewrite it to include some point where parameterisation by $\beth$ may be expressed.

3.4 Aspect-Oriented Programming

Aspect-oriented programming(Kiczales 1997) is a popular solution to level 2 reuse problems which has appeared in some object-oriented languages — most notably as a decoration to mainstream object-oriented programming languages such as Java and C++. It provides clear native mechanisms for solving reuse level 2 problems as presented in the previous section. In this approach creator B is allowed to create a symbolic expression known as a *joinpoint* to name the point in As design where β is referred to. A further expression known as *advice* encodes the modification of the design where β is substituted by β'. AOP addresses the level 2 reuse scenarios we've just discussed. But there are other scenarios, of what we will call level 3 and level 4 reuse, for which it fails.

As we will argue, the key limitation of AOP in these further scenarios is that the aspects encoding joinpoints and advice can't be expressed in the base language. This means that modifications of these parts of a design can't be expressed using joinpoints and advice, but require something new. That is, in the terminology where we present our new open/closed principle in section 5, the space of AOP expressions *fails to be closed*.

4. More Demanding Reuse Scenarios - Levels 3 and 4

The simple scenario in section 3, solved by AOP, COP and similar formalisms, only represents level 2 reuse (with classic Meyerian inheritance solving the 1st-order scenario). In practice, much more demanding scenarios arise quite regularly. For example

- Level 3 reuse scenarios involve two creators, B and C, both wishing to modify the same part (e.g. β) of A's work. While one response to this situation would be to require that B and C create completely separate versions of A's work, this is clearly less desirable than an approach that allows these differing changes to be managed within a single framework. Thus a reuse facility for level 3 scenarios needs some way to manage multiple, potentially conflicting, modifications of the same structure. We will return to this requirement in our section 8 on freely dimensioned context awareness.
- Level 4 reuse scenarios involve the location of to-be-changed βs within A's work. If β occurs inside many layers of structure, introducing a template or other parameterisation point to support the modification will require a good deal of rework. Worse, if there are several βs in A's work, but only some of these should be changed, there may be no suitable point at which one can introduce a template. New facilities are needed to respond to these situations.

The difficulty of addressing these scenarios with current approaches creates distinctions within a population of creators, and thus, *horizons* beyond which the graph of creators cannot grow. Some creators can restructure A's work so as to enable the modifications they want. But other creators won't have this privilege. Even if they might in principle earn the right to make such modifications, as in an open source project, in practice they may lack the resources to do so. Thus the graph of creators fails to be *open* if a language system can't address each level of these reuse scenarios.

5. A New Open/Closed Principle

Here we present a central motivating principle which summarises the complete intent behind a system which addresses all the reuse scenarios in the previous sections, as well as delineating the boundary of a much wider category of scenarios.

To recap, many of our reuse scenarios, especially those seen at Level 3, require resolution of multiple sources of changes competing to modify the same site. A language of changes therefore should be closed, so that no new facilities need be added to enable it to work on itself. When we fail to meet a reuse scenario, after perhaps repeated successful examples of reuse meeting lower-level scenarios, we create a closed "horizon" in our graph of creators beyond which it cannot grow. Therefore:

> What we seek is a *closed algebra of expressions* which will enable an *open graph of creators*.

A mostly equivalent formulation of this principle, from a slightly different point of view is:

> Any expression from a member of the graph of creators can have its effect replaced (or removed) by the *addition* of a further expression from any other member.

To meet these principles, we want to work with an algebra of programs that is closed under difference. That is, given any two programs, α and α_1, that are similar in intention and expression, there should be a third program, δ_1, such that combining α and δ_1 produces a program that is identical in behavior (and close in its expression) to α_1.

5.1 The program addition operator $\oplus$

We might write mathematically, describing the scenario of the previous section,

$$\forall \alpha, \alpha_1, \exists \delta_1 \text{ s.t. } \alpha \oplus \delta_1 = \alpha_1' \simeq \alpha_1 \qquad (1)$$

where $\simeq$ represents two programs with the same behaviour, and $\oplus$ represents the *program addition operator* which is used by creators in any network to combine programs together. Note that $\oplus$ is rarely defined as part of a language definition, since its use appears at the tooling level of a system. For example, in a compiled language, $\oplus$ requires the addition of command-line arguments to the compiler, specifying source files which should be compiled together, whereas in JavaScript written for the web, $\oplus$ requires the specification of `<script>` tags at the head of the page referencing JavaScript source files which should be fetched and interpreted. It's a prerequisite for meeting the openness requirement that the system's facility for addressing $\oplus$ should be available with respect to the particular form in which a program is delivered to a creator in the network — not likely the case if they are delivered in an executable binary form[4].

(Gabriel 2002) observes that an important schism that has opened up in the community is between those working on "systems" and "languages". We observe that it will be impossible to meet the highest levels of reuse in languages which maintain this separation between semantics (studied by language theorists) and runtime behaviour (measured by the systems community). The protrusion of the program addition operator $\oplus$ outside the space traditionally considered interesting by language theorists is an important evidence of this. Similar to CLOS, (Gabriel 2002)'s paradigm example,

[4] It is this additive operator which gives rise to the subtitle of our paper, promising "additive supports"

a description of Infusion, our proposed non-language, will be impossible without also describing how to observe and influence a particular running system containing a program written with it[5]

5.2 Relationship to diffs and patches

The principle implies an unusual characteristic for the language system we are interested in, which is usually reserved only for artefacts processed by the tooling systems which work on them, such as version control systems. The difference between two valid programs is typically named a *diff* or a *patch* in such systems, and is hardly ever a valid program in its own right. What we seek is a language or dialect in which representatives of such differences can be moderately compactly and validly encoded within the language itself.

5.3 This property cannot be provably or fully satisfied

The property we seek is not susceptible to perfect verification or definition. Not all differences among programs need to, or can, correspond to valid programs. Rather, the aim is that the majority of changes creators actually want to make should correspond to valid programs, and that these programs can be found without undue effort.

Thus satisfaction of our open/closed principle could only be verified by a real community creating software artefacts in the pursuit of real ends. To be more clear adequate satisfaction of this principle could not be proved from an axiomatic basis. It could only be experienced in practice.

Can a useful property be so resistant to strict formalization? Consider homoiconicity, the property of a programming language in which the program structure is similar to its syntax. This is also a "soft" property: any language could be said to have it to at least some extent, LISP strongly and C very weakly. The notion is useful despite its not being crisp. The property is also not unrelated to the one we seek — some measure of homoiconicity is clearly essential in a system capable of encoding program differences as programs.

5.4 Relationship to Meyer's Principle

Meyer's open/closed principle is a good foundation for ours. We believe strongly in its primitives and ends — especially in the possibility that an expression may be "closed" in the sense that it may be "closed over" by further creators as a result of being "relatively constant". This allows a form of "referential transparency" in design — the use of the name of an implementation unit can be safely substituted for its referent, allowing for the possibility of caching, memoisation, etc. and similar desirable affordances. We see two fundamental limitations within Meyer's principle:

The need to account for composite structure in the reused artefacts Meyer's formulation only refers to a single artefact at a time as being open or closed. As we discussed above in section 3, reuse scenarios can involve changes to multiple things in a wider aggregate.

The need to account for repeated reuse Meyer's formulation only considers a single exercise of the faculty of reuse. In practice, creative networks spread wider, and the action of reuse should not degrade the potential for further reuse by more distant creators. This leads to our reformulation of the nature of *openness*.

6. Fluid's Infusion System

In this section we will describe the design and motivation of the *Infusion* configuration system, which has been under development in the *Fluid* community for some years. Many of Infusion's features were designed around the need to meet the open/closed principle[6], but it also aims to meet many other needs (dataflow programming, live programming, multi-paradigm collaborative authoring, etc.), which will not be described here.

We hesitate to name Infusion a *language* since it has been explicitly designed to omit several of the characteristics considered traditional amongst programming languages — most notably that of being *Turing complete*. Our favorite designation for the class in which Infusion fits is an *integration domain* (Kell 2009). We'll return to a discussion of some of these omittied characteristics of a language in **??**. In fact, Infusion attempts to attack the the intractible space of language design by factoring the problem — the comprehended part of the problem is described within Infusion's configuration system, expressed in a dialect of JSON, and the uncomprehended part, still requiring the expression of arbitrary progamming language code, is left behind in the *base language* which is currently JavaScript. As the design of Infusion progresses, the balance shifts in favour of the former.

6.1 Infusion's problem domain

Infusion itself is not designed to permit the expression of arbitrary computations, and so there is by design a large variety of tasks to which it is unfitted. It could not be used to write a compiler, an operating system, or indeed even itself. The design space of Infusion is the space of *user programs* — those which mediate some access to state on behalf of an end user, an ordinary member of society, through some form of user interface, most typically a visual one. Paradigm examples of this class of application are office applications or web applications. We argue that what users require from such applications is not *computation* as traditionally conceived, but

[5] Further, following (Gabriel 2002), we observe that Infusion itself is a clear example of the engineering process preceding the scientific one, whilst still proceeding on a principled basis

[6] In fact, the open/closed principle only emerged partway through the design process of Infusion, as it became clear that a fundamental flaw in an older implementation (the one described in (Basman 2011)) was that while apparently meeting all four levels of reuse, it still contained user expressions, *demands blocks*, which could not effectively be additively rewritten by others.

```
fluid.defaults("examples.minimalGrade", {
    gradeNames: "fluid.component"
});

var minimalInstance = examples.minimalGrade();
minimalInstance.destroy();
```

Table 2. A minimal Infusion program

```
HTTP PUT /defaults/examples.minimalGrade {gradeNames: "fluid.component"}
HTTP POST /components/minimalInstance {type: "examples.minimalGrade"}
HTTP DELETE /components/minimalInstance
```

Table 3. A minimal Infusion program issued over the Nexus HTTP protocol

rather, coordinated access to some state in an appropriately context-dependent way.

6.2 The constituents of an Infusion program

Designs expressed in Infusion are structured, at runtime, into a single-rooted tree of implementation units (instances) named *components*. Each component takes its nature from one or more definitions named *grades*. A *grade* is a block of JSON configuration with a globally namespaced name. With a loose analogy, components and grades can respectively be corresponded to the objects and classes of an object-oriented design. We have chosen different names for our units to avoid confusing Infusion users with the very different behaviour and affordances of their equivalents from OO. Table 1 shows a correspondence between some Infusion terms/primitives and some analogous equivalents in traditional usage, together with some commentary on the differences in intention and expected benefits of the Infusion primitives.

6.3 A minimal Infusion program

Figure 4 shows a minimal Infusion program. It registers a grade named `examples.minimalGrade` derived from just the core framework grade `fluid.component`, constructs an instance of it, and destroys it.

This example might suggest that the usage of Infusion is necessarily tied to the use of the base language (JavaScript) but this is not necessarily the case. By means of the *externalization* provided by, for example, the **Nexus** component implemented as part of the GPII's Prosperity4All Program, all of the facilities used in 4 could be addressed from outside the process using standard HTTP endpoints. For example, the following sequence of HTTP requests would have the same effect as in listing 4:

6.4 A little Infusion program showing context-based reference

We move to a slightly higher level of complexity in order to exhibit how Infusion's context-based reference resolution system functions. We will construct a small tree of three components, the root, a child, and a sibling, and a reference

```
fluid.defaults("examples.refRoot", {
    gradeNames: "fluid.component",
    components: {
        child: {
            type: "fluid.component",
            options: {
                siblingValue: "{sibling}.options.value"
            }
        },
        sibling: {
            type: "fluid.component",
            options: {
                value: 42
            }
        }
    }
});

var that = examples.refRoot();
// Next line logs: "Resolved value from child is 42"
console.log("Resolved value from child is ", that.child.options.siblingValue);
```

Table 4. A small Infusion example showing reference resolution

from the child to a value held by the sibling. This shows Infusion's *structural scoping* model.

An expression of the form `{sibling}.options.value` is known as an IoC reference, named after Infusion's role as an Inversion of Control framework. The portion `{sibling}` of the reference is the *context expression*. In this form of reference, this matches upwards through the tree of instantiated components, looking for any parent or sibling of a parent matching the context name. A context name matches in three cases:

- It matches any full grade name that the component is derived from
- It matches the last path segment of any grade name that the component is derived from
- It matches the component's member name with which it is embedded in its containment parent

In our example, it is the 3rd rule which causes the reference to match the sibling on the member name `sibling`. After the context part of the reference has matched, the remainder of the reference, e.g. `options.value` is resolved by sequential property access on each path segment.

The aim of Infusion's resolution system is for all references of this kind to resolve, even if from a conventionally object-oriented point of view they would result in a circular graph of references with respect to constructing objects. The Infusion runtime instantiates an entire component tree as part of a single, data-driven process, and only rejects graphs which are cyclic with respect to individual leaf values.

6.5 Further Information on Infusion

There is not space here to cover many features of Infusion, but comprehensive documentation is available at (Infusion 2016) and it has been described previously in (Basman 2015) which includes a treatment of its support for networks of creators collaborating on a simple application, and (Basman 2011), an older paper which describes some obsolete features but includes a rationale for configuration systems promoting end-user design.

Term	Correlates	Distinction and Similarities	Intention and Advantages
Grade	Type/Class	Rather than establishing *contracts* or describing *storage*, a grade is a block of (JSON) configuration with a globally-qualified name which is merged in an aligned way with others to produce a description from which component instances can be built. Grade names can also be used as *landmarks* (*context names*) in order to bind segments of IoCSS selectors.	The use of grade-based descriptions reduces *excess intention* in descriptions of parts of implementations. The run-time structure of an instance is much more closely tied to the authoring-time structure, allowing for the “notation” of authors and users to be directly corresponded.
Component	Object	Components are instances of grades, as objects are instances of classes. Rather than intending to *insulate* the implementation as a private implementation detail, the purpose of a component is to expose its contents (state, constants and tree of subcomponents) as directly and transparently as possible	The globally visible component tree in an Infusion system is the address space in which the design intentions of multiple authors can be expressed and coordinated. Options distributions could not be expressed if the space of their selectors could not be based on the already transparent address space of the component tree
Invoker	Method	Functions attached to Infusion components are coded for by declarative configuration defining *invokers*. Rather than the *dispatch* model used in object-orientation, where the identity of a requested member function (method) is computed dynamically based on the context of the caller, an execution of a particular invoker will always bind to the same function.	Static dispatch aids performance and clarity in a design, as well as correlating the behaviour of part of a design with that held in another system (an avatar) which may be based on different technologies. All the required benefits of context-awareness may be achieved by reinstantiating the part of a component tree affected by a contextual change.
Model	Model (MVC Programming) / Model (Model-based development/MBD) / Behaviour (Functional Reactive Programming/FRP)	Infusion *models* encode mutable state in a JSON-equivalent form. Taken together with the associated model relay rules, these can constitute a model from the MBD point of view, since the space of model states can be deduced. Finally, the stream of values of a model over time can be compared to an FRP *behaviour*, transduced into other streams via transforming relay rules.	Similar to the use of grades, Infusion models minimise *divergence* between run-time and authoring structures. They also aid liveness and transportation of applications — it should be possible to effectively move an application between systems or users by transmitting just its models.
Options Distributions	Advice (AOP)/Diff (VCS)	An options distribution, like an aspect-oriented programming “advice”, allows an existing application (component tree) to be modified by an author from the outside - that is, they can derive a variant application without modifying the expression of the original author. Unlike an advice, distributions have the same structure and syntax as ordinary configuration.	Since options distributions form a closed system, it is clear how multiple authors can collaborate on the same system, and multiple modifications competing to target the same piece of the design can have their relative priorities negotiated. This also implies that the same authoring tools can be used to write and check distributions as well as ordinary configuration.

Table 1. Guide to terms used in this paper and relation to common forms

7. Further Infusion Topics

In this section we’ll cover some aspects of Infusion’s design which may initially appear extrinsic to the core topic of reusability. Whilst it is possible to imagine alternative systems which meet our reusability principle with divergent viewpoints on these topics, we will produce some indirect arguments which relate these design properties back to the core topic. These areas are:

Avoidance of Computation - We prohibit general computations from being expressed in Infusion’s core, in aid of liveness and transparent authoring

Externalisability - Infusion artefacts and state can be externalised naturally and directly — aiding cooperation with artefacts in other languages and processes.

Related to both of these areas comes Infusion’s lack of *dispatch* — Infusion does not have any concept of a *slot* or any kind of computed property access as is considered essential in most lineages of OO, especially those descended in the Smalltalk lineage. An Infusion component simply has concrete *members*, properties whose value at any time is straightforwardly derivable from looking into state by means of the base language’s member access operator.

7.1 Infusion and Computation

Infusion has been designed to be incapable of computation per se — with a design goal that it fails to consume time and space greater than $O(n \log n)$ when given input of size n[7].

Infusion is explicitly designed to preside over elements of code written in a *base language* (and so has some of the characteristics of a *library*) which in our current implementation is JavaScript, but may in practice be any traditional programming language which allows for the expression of free functions which are “morally” side-effect free. Our aim is to steadily increase the power of the overlying Infusion system at the expense of the expressive power of the base language — which we hope to also impoverish, for a wide range of use cases, below the threshold allowing it to qualify as a progamming language by virtue of being Turing complete.

7.2 A “good function”

We hope to isolate, amongst the base language, a dialect capable of expressing only what we term “good functions”. A good function

[7] The current implementation fails to meet this criterion by a ways, probably consuming $O(n^3)$ time and $O(n^2)$ space or so, but we hope/believe only through faults in implementation rather than underlying design or strategy errors. The point remains that with any polynomial bounds on its resource usage, it fails to qualify as a traditional programming language

- Is a pure function (free of side-effects on the environment)
- Can consume resources no greater than $O(n \log n)$ when given input of "size" n
- Can only apply the control primitives of *conditional execution* and *structural recursion* — that is, control structures such as generalised looping (`for` or `while` loops) or arbitrary recursion do not occur

Whilst arbitrary progamming language structures *can* appear in the base language, we would like to highlight them during the authoring process as a form of *taint* — similar to the way in which Perl language elements may be declared *tainted* through having processed unchecked user input data, or C# language elements may be considered *unmanaged* as a result of having accessed machine resources such a memory or threads in an unchecked way.

These taints may infect the authorial process in a variety of ways. Firstly, if a base language expression is tainted through possible excess resource consumption, it might impact the *liveness* of the authoring process, which we would want to remain highly responsive in typical situations. Secondly, if an expression is tainted through committing side-effects, it would interfere with the authorial process by taking the user/designer through transitions which could not easily be reversed. Both of these kinds of taints mark out a code block as being suitable for being provided with accompanying *mock* or *null* implementations which would abridge its operation in typical design situations.

7.3 A New Cellular Model

The organisation of a Smalltalk application into insulated units named "objects" was inspired by the subdivision of biological entities into cells (Kay 2013). This is good engineering for systems which must be self-assembling and self-managing, but is a poor fit for systems which must place all of their resources for adaptibility at the disposal of the user — or a wider network of creators. Our cellular units are named "components", and rather than serving to insulate parts of the implementation one from the other, they serve the converse end of maximally advertising the structure of the application via a transparent addressing scheme. Infusion components have a further vital role in structuring an application, in that their lifecycle points are used to structure the lifetimes of relationships and adaptations in the component tree.

Our inspiration is taken from a very popular and successful idiom for end-user programming — the Document Object Model (DOM - (W3C 2002)) mediating access to the rendered contents of web pages. A crucial affordance which has emerged from applications based on the DOM is the use of CSS selectors to stably represent selections of the tree of DOM nodes. The original use case for CSS selectors allowed designers to target styling rules at parts of a web interface, which rules could expect some stability of reference as the content was designed. Over time, as web interfaces became more dynamic, CSS selectors became a vital part of the implementation design as well, as mediated by popular frameworks such as jQuery.

Our cellular model, thus, imports two vital elements from the idiom of DOM-based programming:

7.3.1 Transparent, selector-based addressing

A selection of tree nodes which is to be targetted with some effect or predicate can be stably identified by means of a pattern encoded into a string, with clauses representing intermediate match sites in the tree combining to represent the final match. In Infusion, our selector dialect is ***IoCSS***, named after one of the framework's original roles as an "Inversion of Control" system. It is structured very similarly to the CSS system, only with a greatly reduced set of predicates and combining rules.

7.3.2 Coordinated lifecycles with peers

The DOM is an environment where elements may unpredictably come and go. It's crucial for application integrity that any effects associated with the existence of a node are banished along with its demise. As well as simple examples such as event handlers attached to a DOM node itself, there are more subtle possibilities such as programming language (JavaScript) code which has "closed over" a reference to a DOM element which has been destroyed. It's crucial that the system behave gracefully in such mixed authorial scenarios.

In Infusion, there are yet more complex possibilities of multilateral relationships amongst component nodes. For example, one component may bind an event listener on behalf of another, set up a dataflow relationship between itself and other components, or broadcast options distributions into the tree at large. All of these relationships must be cleanly torn down when the component is destroyed.

The lifecycle of components also provides crucial landmarks in *time* whereby the scope of dynamic adaptations can be demarcated. We will see examples of this in our worked context awareness example in section 8.1.

8. Freely Dimensioned Context Awareness

In this section we will rework an example from Ungar et al's Korz system(Ungar 2014) which demonstrated how fresh "dimensions" of adaptibility can be contributed into a target artefact from multiple sources. This represents a high-order case of reusability (in terms of section 4, and exhibiting the use case in two systems will shed light on both systems as well as on the nature of the problem space.

Despite both being rare examples of systems in which such a high-order reuse case can be met, the architectures of Korz and Infusion are extremely different. To start with, Korz can function as a general-purpose programming language, whilst Infusion cannot. Further more, the differences in the *dispatch model* of the two systems are profound. Korz is a language with highly dynamic dispatch, descended in a direct lineage from Smalltalk via the Self language, inheriting these languages' conceptions of "slots", named entries within an implementation unit (an *object*) where a runtime computation occurs in order to locate a particular concrete implementation in response to a message. In contract, Infusion has no dynamic dispatch whatsoever — the dispatch choices to be made by an implementation unit (a *component*) are built into it at its point of instantiation. As we will see as we elaborate our example, this lack of dynamic dispatch does not limit the dynamism a runtime Infusion system, and in fact makes it easier to quantify and bound this dynamism and hence export it into other environments. In our example we will show how the dynamic content of part of an Infusion component tree can be easily exported into an environment traditionally very hostile to dynamism, the implementation of a WebGL shader operating a live filter of a video stream, written in the GLSL shader language.

8.1 Korz Adaptation Example

The example presented in (Ungar 2014) demonstrates how fresh adaptations can be contributed to a target implementation, without either a change in its implementation or a change in the type name consumed by its users. This represents a modern, high level of adaptibility, which is also present in such environments as Newspeak (Bracha 2010). We will work through this example using Infusion's context awareness facility. [TO APPEAR]

9. A Real-World Example of Type 4 Reuse

The Global Public Inclusive Infrastructure (GPII) is an ambitious project whose aim is to implement an auto-personalisation system which makes the resources of operating system and application-level adaptation available to users across all applications and platforms. A core architectural component is the *flow manager* which coordinates the workflow of assembling information about the user's needs and preferences, the capabilities of the local device and relevant privacy policies, and orchestrating the capabilities of the device to bring it to an inferred condition meeting the needs and preferences. As such it represents a moderately deeply nested containment structure of the types we have discussed in 4 in which more advanced reuse requirements have arisen in real practice. However, we should all the same stress that this still represents an architecture only at modest rather than extreme scale, currently comprising thousands rather than millions of lines of code. Therefore we feel justified in positioning even level 4 reuse as a standard, everyday level of reusability that every competent architecture should aspire to.

In table 5 is a section of configuration for a "driver" of this system, meeting the needs of a particular integrator who wishes to make choices about configuration of a storage engine in one of the several deployment scenarios of the application, one where a particular set of components of the flow manager are deployed locally on the user's local machine, and others are deployed remotely "in the cloud" (other such scenarios position these component differently — for example, all locally, or almost all remotely, etc., this flexibility of deployment scenario being one of the main drivers of the requirement for reusability.

On line 12, we see an IoCSS selector with 4 components `{that cloudBasedConfig flowManager preferencesDataSource}` selecting by path a particular subcomponent of just *one* instantiation of a `flowManager` component for "advice". This full specification is necessary because in this configuration, *two* such instances exist along different containment paths, representing the traditionally local and remote functions of the system, and we wish to advise only one of them. This clearly represents level 4 reuse, since in any traditionally constructed system, the "substitutability" of these two instances via their common inheritance from a base grade (in this case, `gpii.flowManager'`) would render them indistinguishable from the point of view of a consumer of the design.

10. Conclusion

We have presented a tower of increasingly sophisticated scenarios of reuse, stretching from classical object-orientation's reuse at level 1 as "Meyerian Reuse" up to more demanding requirements characterised as level 4, which we nonetheless argue represent everyday levels of reuse that arise frequently in real architectures (whether characterised or not). We have assigned popular language and tool idioms to particular levels of this tower according to the sophistication of reuse requirements which can be natively met by their means, showing that no popular systems are capable of meeting the requirements of level 4 reuse. We have argued that fundamentally different architectural strategies, those informing our design of Fluid's Infusion system, are required to meet such requirements, and that several other "quality of implementation" issues such as straightforward interactions with external systems implemented in different processes, languages and idioms, are also met by them.

We have produced a compact new principle, our open/closed principle, which summarises the requirements of all 4 levels of this tower in a straightforward, transparently motivated statement, as well as encompassing a much wider terrain of as yet unarticulated reuse requirements. We explain that meeting this principle perfectly is impossible even in principle, but instead represents a constant cycle of implementation, self-observation and refinement that will steadily increase the terrain of expressions which can usefully be subsumed under the principle in a realistic space of user designs. We devote ourselves to this struggle.

```
{
    "type": "untrusted.development.all.local",
    "options": {
        "gradeNames": ["kettle.multiConfig.config"],
...
        "distributeOptions": {
            "untrusted.development.port": {
                "record": 8088,
                "target": "{that cloudBasedConfig}.options.mainServerPort"
            },
            "untrusted.development.prefs": {
                "record": "http://localhost:8088/preferences/%userToken",
                "target": "{that cloudBasedConfig flowManager preferencesDataSource}.options.url",
                "priority": "after:flowManager.development.prefs"
            },
...
    }
}
```

Table 5. Example of GPII FlowManager configuration showing resolution of level 4 reuse

References

Basman, A. et al. *Harmonious Authorship from Different Representations*, Proceedings of the 26th Annual PPIG Workshop, 2015

Basman, A. et al. *To inclusive design through contextually extended IoC*, Proceedings of the ACM OOPSLA Companion (Wavefront), 2011

Bracha, Gilad et al. *Modules as Objects in Newspeak*. Proceedings of the 24th ECOOP, June 21-25 2010. Springer Verlag LNCS 2010.

Le Hégaret, Philippe. ”The W3C Document Object Model (DOM)”. World Wide Web Consortium, 2002 `http://www.w3.org/2002/07/26-dom-article.html`

Gabriel, R.P. *The structure of a programming language revolution*, Proceedings of the ACM Onward 2012, pages 195-214. Springer NY 2012.

Fluid Infusion Documentation `http://docs.fluidproject.org/infusion/development/`

Kay, Alan *“E-Mail of 2003-07-23”. Dr. Alan Kay on the Meaning of “Object-Oriented Programming”*. `http://www.purl.org/stefan_ram/pub/doc_kay_oop_en`

Kiczales, G. et al. *Aspect-oriented programming*. Proceedings of the 11th ECOOP (1997).

Kell, S. *The mythical matched modules: overcoming the tyranny of inflexible software construction*, Proceedings of the 2009 OOPSLA, pages 881-888, ACM.

Klokmose, C. et al. *Webstrates: Shareable Dynamic Media*, Proceedings of the 2015 UIST, pages 280-290, ACM, New York.

Meyer, Bertrand. *Object-Oriented Software Construction*, Prentice-Hall, 1988

Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

Ungar, D. et al *Korz: Simple, Symmetric, Subjective, Context-Oriented Programming*, Proceedings of the Fourth Symposium on New Ideas in Programming and Reflections on Software (Onward), ACM, 2014