The Open Authorial Principle

Supporting Networks of Authors in Creating Externalizable Designs

Name1

Affiliation1

Email1

Name2   Name3

Affiliation2/3

Email2/3

Abstract

We introduce a new principle, the *open authorial principle*, that characterises desirable properties of languages and configuration systems supporting networks of authors. We survey the growth in generosity of authorial systems, 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 examples originally developed for Korz and Newspeak into Infusion's configuration system, and discuss how multiple authors can additively contribute fresh implementation dimensions into the same artefact, and how to resolve priority amongst their contributions. We show a working system that allows adaptations to be dynamically contributed into a video processing pipeline, and shed light on how systems may be designed to facilitate the provision of "avatars", replicating the function of a segment of an implementation for a bounded time.

Keywords   context awareness, declarative configuration

1. Introduction

Reuse is the capacity of a design to empower others to continue the design process via extension or adaption. Developments in programming idioms increasingly support this capacity in richer networks of authors working on artefacts with more complex structuring. In this paper, we will survey a series of increasingly generous idioms which we will categorise into a 4-level hierarchy according to the sophistication of the reuse stories that they support, starting with object-orientation at the base level 1. Our stories will be structured around authors through whose hands designs successively pass, conventionally labelled A, B, C, etc. in which collection an "end user" E is morally incorporated. We'll assemble these authors into *networks* in which authors who exchange design artefacts are connected by arcs in the network. One example of such a connection is whereby A writes source text that 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.

1.1   Horizons in the network of authors

We consider that all existing programming idioms create unwelcome distinctions among a population of authors, and thus, *horizons* beyond which the graph of authors cannot grow. Some authors can restructure the work of an originating author to enable the modifications they want. But other authors 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 authors fails to be *open* if a language system can't economically address each level of these reuse scenarios — that is, to deliver the affordances of reuse at a cost that the interested community can afford. We'll consider that a system *de facto* fails to deliver these affordances economically if it incurs costs amongst the authors that grow much faster than linearly with respect to their number and the size of design they're collaborating on.

As we progress through increasingly sophisticated levels of reuse, we will observe that the horizon bounding the graph of authors is steadily pushed back. At level 2 we will meet developments such as Aspect-Oriented Programming (Kiczales 1997) and Context-Oriented Programming (Costanza 2005), and at level 3 more modern and ambitious systems such as Newspeak (Bracha 2010) and Korz (Ungar 2014). In order to situate the entire hierarchy of levels, showing the way to level 4 and beyond, we will summarise our conception of reuse under a central organising principle, our new *open authorial principle* (OAP). This principle provides a light in which the potential for any design idiom to promote generous reuse can be clearly assessed. The principle, elaborated in section 5, states

Principle (The Open Authorial Principle). *Any expression by one author should allow its effect in the design to be replaced by an additional expression by a further author.*

1.2   An algebra of program differences

The principle implies an unusual characteristic for the language system we are interested in, which is usually reserved only for artefacts as processed by the tooling systems that 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 fairly compactly and validly encoded within the language itself.

This goal gives rise to what we will call an *algebra of program differences* in which expressions are combined. In section 5.1 we will informally consider a *program addition operator* $\oplus$ combining members of the algebra. This operator lies outside the space traditionally considered part of a programming language design. Our reuse goal implies refining the action of the operator under which program differences are combined. Our algebra cannot be operated with crude, unstable source-level coordinates such as program line numbers directing where differences are taken from and where they are projected to. We need to recast the work of programming in terms of a *natural coordinate system* in which units of design have meaningful, stable names which identify their location in a fine-grained tree of cells (section 7.1) within the design. This work has underlain the design of our system, Infusion, which offers level 4 reuse in our hierarchy and will be described in section 6.

1.3 Reuse levels

The following sections will tour our hierarchy of reuse levels, an overview and illustrations of which appear in table 1. In this presentation of reuse levels, we'll try to use the term "implementation unit" rather than idiom-specific terminology such as "object", "class", "module", "type" etc. for naming units, to avoid biasing the discussion.

2. Meyerian Reuse - Level 1

Meyer's open/closed principle (Meyer 1988)[1], codifies what is now accepted as one of the core principles of object-orientation. Meyer's principle allows for what could be described as "first-order reuse". One author can use the definition of an implementation unit to derive an elaborated definition without requiring the original author to modify their definition. This provides only for reuse of single implementation elements at a time (classes/objects), and does little to facilitate reuse across a design or of larger aggregations — in fact, we will argue that object-orientation actively impedes such wider reuse.

In Meyer's presentation[2], there are implicitly authors A, B, and (at least one) end user E. 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 feature of an object-oriented language granting the ability to create an artefact taking the

form of a *base class* or *interface*[3] $\aleph$ (aleph) which expresses some of the 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. This situation is illustrated in row 1 of table 1. Note that this swap can only be made 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 involve a variety of "factory pattern"(Gamma 1994). We will return to these 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. Author A has created an implementation unit named α which contains a subunit named β. Author 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 author E as a substitute for α. This situation is illustrated in row 2 of table 1.

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 explicitly constructed 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 a "factory pattern" — the creation of an intermediary artefact we might name $\beth$ (beth) whose purpose is to abstract over the construction point of β. This is obviously unsatisfactory[4] 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. (Bracha 2013) has commented on this form of pathology of "design regress", observing that it results when a "shadow world" is created as a result of resolving a problem with design artefacts that don't have a first-class design status. Other responses to this problem require a fresh language feature, orthogonal to the classically object-oriented ones, al-

---

[1] To paraphrase, "A module should be available for extension (*open*) but also 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.

[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.

[3] The presence of $\aleph$ is actually a detail arising from (Martin 1996)'s representation of the OCP, but the two presentations are identical from the point of view of their authorial affordances for the OAP.

[4] "*entia non sunt multiplicanda praeter necessitatem*" (entities should not be multiplied beyond necessity) as Occam observed

Level	Example Scenario	Systems Treating	Section	Diagram of Example Scenario
Level 1	A single artefact α created by A is extended to α' by B	Object Orientation	Section 2	
Level 2	A's α with a nested artefact β has β extended to β' by B, creating an overall α'	Parameterised Types, AOP	Section 3	
Level 3	B extends A's α to a collection of α_n, C wants to extend all of α_n by α_C without work proportional to n	Newspeak and AOP (wide hierarchy), Korz (all)	Section 4, Section 4.1, Section 8	
Level 4	A has created an extended containment hierarchy, containing some scattered γ at a deeply nested level. B, C etc. want to extend the entire hierarchy adjusting only some γ to γ', without work proportional to the size of the hierarchy	Infusion	Section 4, Section 10	

Table 1. Table of levels of increasingly sophisticated reuse with illustrations

lowing the expression of "parameterized" or "generic" types — we'll return to this possibility in section 3.2.

3.2 Containment through inheritance

Self-aggregation via inheritance is another strategy for reuse in object-orientation. With inheritance, 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 OO and essentially forces the requirement for parameterised types to be added to the system. Parameterised types allow a definition to be generalised over all values of a type which appears in it.

3.2.1 Reuse through parameterised types

In C++, author 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 author 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 cannot be mentioned without also bringing the requirement to mention the particular $\beth$ it involves.

An elegant solution to this "foresight" problem is comprised in solutions such as (Bracha 2010)'s Newspeak, in which every type name is implicitly parameterised — at every point of consuming a type name, a user has the facility to rebind it to one resolvable in their context. Newspeak resolves not only reuse problems at this level but also some at the following level 3.

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 solution to level 2 reuse problems which has appeared in some object-oriented languages — most notably as a decoration to mainstream object-oriented languages such as Java and C++. It provides clear native mechanisms for solving reuse level 2 problems as presented earlier in this section. In this approach author 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*. More primitive expressions of the same intent behind AOP are available in traditional OOP, under the names of "decorator patterns" or "visitor patterns"(Gamma 1994). However, as well as suffering from the poor compositional properties of AOP[5], these also suffer from the same problem noted against templates in section 3.2.1 — they may not be deployed without design forethought.

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 authors, B and C, both wishing to modify the same part (e.g. α) 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. If B and C have to work with separate artefacts, they have become disconnected in the graph of authors. Thus a reuse facility for level 3 scenarios needs some way to manage multiple, potentially conflicting, modifications of the same structure.

- Level 4 reuse scenarios involve the location of the to-be-changed elements, γs within A's work. If γ 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 γs 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. Such a scenario is illustrated in row 4 of table 1, and exampled in section 10. New facilities are needed to respond to these situations.

4.1 Refining Level 3 Reuse Scenarios

Note that level 3 reuse scenarios may be demanding on account of two orthogonal forces. The first force stresses designs where B has extended A's design into a deep hierarchy, and C wishes to advise all of it. (Bracha 2010) names this variety of level 3 reuse as the "class hierarchy inheritance" problem. In the formulation there, author A has created a base class `Shape`, author B a `ShapeLibrary` deriving `Rectangle`, `Circle`, etc. and author C has created a hierarchy of colorable things, and wishes to make all the contents of B's library of shapes available to E as coloured shapes without having to do work proportional to the size of the library. This situation is illustrated in row 3 of table 1, and we will return to it in section 8 where we work through an example from (Bracha 2010)'s Newspeak. The second force stresses designs where there is a large number of authors, C, D etc. all competing to extend the same artefact. (Ungar 2014) names this variety of level 3 use as requiring "symmetric dimensions of context", and we will return to this scenario in section 9 where we work through an example from (Ungar 2014)'s Korz.

5. The Open Authorial Principle

Our reuse scenarios, characterised from levels 1-4 in the previous section, as well as a wider universe of uncharacterised scenarios, may be generalised by our Open Authorial Principle, as stated in section 1.1:

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

This principle can be looked at from a different point of view in terms of the algebra of program differences mentioned in section 1.2. Many of our higher-level reuse scenarios require resolution of multiple sources of changes competing to modify the same site. The language of expressions, therefore, should give rise to an algebra 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. An example appears in section 8, where α consists of a shape, δ_1 consists of the addition of a colour, and the resulting α_1 represents a coloured shape. Should our language fail to meet a reuse scenario, we create a closed "horizon" in our graph of authors beyond which it cannot

---

[5] It is hard to decorate a decorator, since it has no clear coordinates in the design — all one can do is add oneself to the list of decorators attached to the same base artefact.

grow. Therefore an alternative statement of the OAP, which we'll elaborate in the next section, is as follows:

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

5.1 The program addition operator $\oplus$

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

$$\forall \alpha, \alpha_1, \exists \delta_1 \text{ s.t. } \alpha \oplus \delta_1 = \alpha'_1 \simeq \alpha_1 \tag{1}$$

where $\simeq$ represents two programs with the same behaviour, and $\oplus$ represents the *program addition operator* which is used by authors in any network to combine programs together. Note that $\oplus$ is rarely defined as part of a language definition, since its use more usually 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 is 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 author in the network — not likely if it was delivered in an executable binary form[6].

(Gabriel 2012) observes that an important schism has opened up in the community 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 2012)'s paradigm example, a description of Infusion, our proposed language/system, will be impossible without also describing how to observe and influence a particular running system containing a program written with it[7].

5.2 This property cannot be provably or fully satisfied

The property we seek is not susceptible to perfect verification, because it establishes properties observed by real users in real communities — as explained in section 1.1, the principle's subject matter is the economics of authorship rather than axiomatised theory. Not all differences among programs need to, or can, correspond to valid programs. Rather, the aim is that the majority of changes authors actually want

to make should correspond to valid programs, and that these programs can be found without undue effort.

Can useful properties lie outside the domain of axiomatised theory? 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.

5.3 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 authors 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 elements 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 authors. 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 to support the OAP[8], but it also aims to meet many other needs (dataflow programming, live programming, literate programming, multi-paradigm collaborative authoring, etc.), which are outside the scope of this paper.

We hesitate to name Infusion a language since it has been explicitly designed to omit several characteristics considered traditional amongst programming languages — most

---

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

[7] Further, following (Gabriel 2012), we observe that Infusion itself is a clear example of the engineering process preceding the scientific one, whilst still proceeding on a principled basis.

[8] In fact, the OAP 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.

notably that of being *Turing complete*. Our preferred designation for the class in which Infusion fits is an *integration domain* (Kell 2009). Infusion attempts to attack the the intractable 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 programming language code, is left behind in the *base language* which is currently JavaScript[9]. As the design of Infusion progresses, the balance shifts in favour of the former.

Infusion is used on a daily basis by software teams meeting real ends, and has comprehensive documentation available at (Fluid Team 2017).

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 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 the related constructs in OO. Table 2 shows a correspondence between some Infusion terms/primitives and these related concepts, together with some commentary on the differences in intention and expected benefits of the Infusion primitives.

---

[9] A misunderstanding of the nature of the OAP might allow one to pose the argument that basing Infusion on a dynamic language such as JavaScript substantially satisfies the OAP automatically. This argument is false, since the real economics of authorship don't support it — whilst the in-memory representation of objects in a dynamic language is quite plastic, this goes little way to helping users to find relevant objects at runtime and adapt them, preferably before their immediate clients start to consume them – see section 3 for examples. In practice, the economics of adaptation for ordinary dynamic languages run on the same basis as those for static languages — locating the source code for the artefact, forking and modifying it.

Infusion grades more closely resemble "mixins" seen in some object-oriented traditions (for example, the Flavors/CLOS/etc. lineages discussed in (Gabriel 2012)) than simple classes, since multiple grades may be listed for inheritance at each level of the hierarchy, and each may appear multiple times along the paths to root. An important difference in Infusion is that the results of grade resolution themselves take the form of a JSON document, a component's *merged options*, which are attached in immutable form to an instantiated component's top-level property named `options`. This allows for the results of the merging and expansion process to be interpreted and consumed by other authors and tools in the system rather than simply the system's own runtime, and is an important example of Infusion widening the collection of privileged authors in the ecosystem, beyond the system's own compiler/interpreter, tools and runtime, a development motivated in (Basman 2015).

As well as simply coding for plain values, component options are interpreted by the runtime into specialised elements which are attached to the component on construction, such as events and their listeners, dataflow-driven models and constraints on these, bound functions (invokers), etc. which there is not room to describe here.

Ordinarily, multiple sources of component options are simply superimposed as JSON structures using an algorithm very similar to that seen in many popular utilities, for example, jQuery's `extend` algorithm. However, the user can supply a `mergePolicy` option to modify the process by which particular named options are collected and merged. We will see an example of this in our context adaptation sample in section 9.1.

6.2.1 References and Distributions

A component's merged options reflect more than just the options expressed in the hierarchical resolution process. In addition, options can be pulled in from the surrounding component tree by resolving *IoC references* (as shown in section 6.4) and also received from *options distributions* which allow options to be broadcast (pushed) out to other locations in the tree (seen in section 8). IoC references fill some of the roles of a *linker* in other environments, which allow a design to be assembled from pieces which are then orchestrated into a complete program by "fixing up" symbolic references with respect to a context. Options distributions allow differences between programs to be represented as part of a program, which is the crucial requirement for our reusability principle (see section 5). These references represent the stable design coordinates described in section 1.2, a prerequisite for which is Infusion's encoding within a JSON tree with design-meaningful names as keys.

The aim of Infusion's resolution system is for references 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,

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

```
1  fluid.defaults("examples.minimalGrade", {
2      gradeNames: "fluid.component"
3  });
4
5  var minimalInstance = examples.minimalGrade();
6  minimalInstance.destroy();
```

Listing 3. A minimal Infusion program

data-driven process, and only rejects graphs which are cyclic with respect to individual leaf values.

6.3 A minimal Infusion program

Listing 3 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. Some implications of the explicit destruction lifecycle point triggered by `destroy()` are discussed in section 7.1.2.

This example is expressed in JavaScript, which could suggest that the usage of Infusion is necessarily tied to the use of that language, but this is not the case. By means of the *externalization* provided by, for example, the **Nexus** component implemented as part of the GPII's Prosperity4All Program (Nexus 2017), all of the facilities used in listing 3 could be addressed from outside the process using standard HTTP endpoints. For example, the sequence of HTTP requests in listing 4 would have the same effect as in listing 3.

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

Listing 4. Listing 3 issued over the Nexus HTTP protocol

```
1  fluid.defaults("examples.refRoot", {
2      gradeNames: "fluid.component",
3      components: {
4          monty: {
5              type: "fluid.component",
6              options: {
7                  siblingAge: "{rachael}.options.age"
8              }
9          },
10         rachael: {
11             type: "fluid.component",
12             options: {
13                 age: 42
14             }
15         }
16     }
17 });
18
19 var that = examples.refRoot();
20 // Next line logs: "Resolved value via monty is 42"
21 console.log("Resolved value via monty is ", that.monty.options.siblingAge);
```

Listing 5. A small Infusion example showing reference resolution

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. In listing 5 we construct a small tree of three components, the root and two child components, `monty` and `rachael`, and a reference from the child `monty` to a value held by the sibling `rachael`. This shows Infusion's *structural scoping* model.

An expression of the form {rachael}.options.age as appearing at line 7 in listing 5 is known as an IoC reference, named after Infusion's role as an Inversion of Control framework. The portion {rachael} 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 rachael. After the context part of the reference has matched, the remainder of the reference, e.g. options.age is resolved by sequential property access on each path segment.

7. Addressibility and Externalisability

In this section we'll discuss two closely related properties of Infusion's design which we argue emerge from the reusability principle.

Free Addressibility - Every part of every Infusion component can be referenced using a global path expression, encoding its path as descended from the global component tree root.

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

7.1 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 2003). 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 adaptability at the disposal of the user — or a wider network of authors. 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.

As a result of the DOM's huge currency at the core of the world's web browsers, DOM implementations have become extremely robust and performant platforms for shared authorship of a space of user interface elements, inspiring such implementations as (Klokmose 2015)'s *Webstrates*, a collaborative authoring environment where the state of the DOM itself corresponds to the authorial shared state.

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

7.1.1 Transparent, selector-based addressing

A selection of tree nodes which is to be targeted 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. 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. From the descendant selector rules of CSS we import only the descendant combinator consisting of whitespace, and the direct descendant rule >. For node selectors we allow just one possibility, a string matching contexts according to the rules of section 6.4, and the logical AND combinator &.

7.1.2 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. A typical behaviour to maintain integrity is to in some form "neuter" such a destroyed element, so that it can no longer participate in making side effects visible to the user. In the DOM parlance, it is "detached from the document", and further operations with it remain valid, but can no longer influence the browser's rendering process. In Infusion, we act similarly, and prevent any further event listeners from being serviced on a component which has been destroyed. This is quite at odds with a typical OO approach, in which there is not intended to be any distinct lifecycle state in which an object reference is visible to referrers and in which the object is not considered "live". These lifecycle requirements go beyond those of traditional garbage collection because of the situation where a freshly destroyed node may *imminently* be targeted by an upcoming effect, say an event notification which is upcoming on the call stack.

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 9.1.

7.2   Externalisability and REST

A systematic failure of object-oriented environments is their tendency to be "hermetic" — the semantic is defined in great detail of the behaviour of an implementation within a particular "walled garden" (the language itself and its virtual machine), and only limited thought is given to how this implementation is expected to coexist in a busy mixture of distributed elements written in a mixture of implementation technologies and idioms. The only common model for application distribution in the OO community is the "proxy" model, where a local agent (an "object") is considered to be a proxy for a remote one, fielding local messages, converting them into messages to the remote part of the system, awaiting a response and then issuing that response locally on behalf of the local client of the proxy. This is part and parcel of the "message passing" model of distribution on which object orientation is founded.

This model is sometimes highly appropriate — especially when the messages passed are small and relatively infrequent, and/or the network has high bandwidth, reliability and low latency with respect to the application's requirements. However, it is not appropriate for applications where the throughput of such messages would be extremely high, or the application is extremely widely distributed over a collection of nodes joined by a network which is neither hugely reliable or capacious.

As we identified in section 7.1, the web is a highly evolved and successful emergent architecture devoted to solving the problems of distributed application development, although it is frequently not recognised as such by computer scientists. The DOM idiom that we praise in section 7.1 is part of a wider engineering idiom named REST by (Fielding 2000). In this idiom, the response to a remote endpoint is not merely a message responding to an arbitrary query, but an exhaustive summary of the state of a *resource*. The acronym REST denotes *representational state transfer*, indicating that *state* is moved from place to place, rather than merely the answers to limited questions as with message passing. Infusion, similarly, places state (and not message passing) at its architectural core and facilitates architectures that work with it.

Transferring application state in bulk (externalising sections of an application) has been noted by some authors as

highly desirable for many authorial tasks — for example, (Kell 2012) notes that several changes in JVM design would be desirable in order to make it more "observable" for debugging purposes.

8.   Infusion's options distributions

Options distributions are the key feature through which Infusion designs enable the solution of level 4 reuse problems. They loosely correspond to *aspects* as seen in aspect-oriented programming systems, in that they allow a part of a design to be marked for a change and "advised" in order to encode the change. As with all Infusion elements, these take the form of JSON-encoded options supplied to a grade. Each Infusion component may accept an options area named `distributeOptions`, which may hold multiple, freely addressed distributions. Each distribution must contain an element `target` encoding an IoCSS selector identifying, relative to the component's current location in the tree, where in the tree the distribution is to be targeted[10]. It must then include either an element `record` holding literal options material to be distributed, or else an element `source` holding a further IoCSS selector signifying that the distributed material should be sourced from existing material in the tree.

8.1   A simple example showing an options distribution

In listing 6 we show Newspeak's level 3 coloured shape example rendered in Infusion (as in the original sample, we'll omit the actual definitions of shapes, colours, etc.)

Here, author C's expression is the component defined at line 16, and wishes to advise all shapes produced within author B's `shapeLibrary` hosted at line 25 that they should have the `examples.coloured` grade mixed in to them. It does this by targeting via the IoCSS selector at line 20, whose effect reads, "For all instances of `examples.shape` nested anywhere below the `shapeLibrary` held at the top level of containment in the overall `colouredShapes` tree, contribute the grade name `examples.coloured` to their grade list".

8.1.1   Options distributions as encoding of "diff"s

It is through options distributions that differences between designs are encoded and transmitted around the system. The two parts of the IoCSS selector held in the `target` field — the context portion, and the path portion — allow the *address* of the modified portion of the design to be compactly encoded. This means that diffs which should be small (for example, those that simply override a single deeply nested

---

[10] The affordances of IoCSS selectors are a little reminiscent of XPath selectors(W3C 2017) as hosted within XML Transformations (XSLT). We head off the unruly implications of such systems for maintenance by two main routes. Firstly, all Infusion configuration is immutable, composited into the running application by a merging algorithm which allows the *provenance* of all final configuration to be tracked back to its source user expression. Secondly, as opposed to XPath which is a programming language its own right, IoCSS selectors permit only the 4 primitives listed in section 7.1.1

```
1   fluid.defaults("examples.shape", {
2       gradeNames: "fluid.component",
3       ...
4   });
5   fluid.defaults("examples.shapeLibrary", {
6       gradeNames: "fluid.component",
7       ...
8   });
9   fluid.defaults("examples.coloured", {
10      gradeNames: "fluid.component",
11      ...
12  });
13  fluid.defaults("examples.colouredShapes", {
14      gradeNames: "fluid.component",
15      components: {
16          colourizer: {
17              type: "fluid.component",
18              options: {
19                  distributeOptions: {
20                      target: "{colouredShapes > shapeLibrary examples.shape}.options.gradeNames"
21                      record: "examples.coloured"
22                  }
23              }
24          },
25          shapeLibrary: {
26              type: "examples.shapeLibrary",
27              options: {
28                  components: {
29                      shape: {
30                          type: "examples.shape"
31                      }
32                  }
33              }
34          }
35      }
36  });
37
38  var that = examples.colouredShapes();
39  // Next line logs: "shapeLibrary's shape is coloured: true"
40  console.log("shapeLibrary's shape is coloured ",
41      fluid.componentHasGrade(that.shapeLibrary.shape, "examples.coloured"));
```

Listing 6. Options distribution in level 3 reuse Infusion sample

value in a single component) have small encodings. For encoding bigger differences, it is more economical for the author to package up a set of related changes to be made in a target component into a (possibly ad hoc) grade, and to distribute that grade name to an entire target component, rather than attempt to account for all the design changes individually. The use of a grade in this case increases the chances that a further author may in the future successfully reuse this difference itself in order to target it to make further changes. We'll see a real-world example of such a grade broadcast meeting a level 4 reuse problem in section 10.

9. Freely Dimensioned Context Awareness

Here we rework an example from the Korz system (Ungar 2014) which demonstrated how fresh "dimensions" of adaptability can be contributed into a target artefact from multiple sources. This represents a high-order case of reusability (level 3 in terms of section 4), and showing it in two systems will shed light on both systems as well as on the nature reusability. Korz is a fertile ground for comparison, since its motivation takes a very similar initial line of argument to that supporting Infusion — observing that object-orientation meets limited needs in contexts where only a single dimension of variation is in play, but needs to be extended to deal with realistic, extended designs involving multiple authors and dimensions. However, the subsequent trajectory of the argument has led to very different systems.

Despite both being rare examples of systems meeting high-order reuse cases, the architectures of Korz and Infusion are extremely different. To start with, Korz can function as a general-purpose programming language, whilst Infusion cannot. Furthermore, 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 and the COP tradition (Costanza 2005), inheriting these languages' conceptions of "slots", named entries associated with an implementation unit where a runtime computation occurs in order to locate a particular implementation in response to a message. In contrast, 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 in our example, this lack of dynamic dispatch does not limit the dynamism of 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 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.

9.1 Context 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 adaptability, which is also present in such environments as Newspeak (Bracha 2010). We will work through this example using Infusion's `contextAwareness` facility (ContextAwareness 2017).

The example application in (Ungar 2014) represents a rendered image with an operation named `drawPixel`, accepting three arguments, x and y coordinates and a colour pixel to be plotted at those coordinates' position. The user on whose behalf the image is to be rendered is considered to have some "context" accompanying them. The image rendering process should be modified by this context, in order to respond to the needs which the context implies. The examples provided in (Ungar 2014) of such contextual requirements include:

- A "colour blind" user on whose behalf the image will be rendered in grayscale rather than in colour

- An "Australian" user on whose behalf the image will be rendered upside down

- A user from Antarctica on whose behalf the image should be rendered at double size as well as upside down

This example was crafted to exhibit that these contexts represent more or less "orthogonal" dimensions of adaptability for the target application, and that they are contributable to the target without interfering either with its implementation or unduly with each other. As examples of interactions (Ungar 2014) consider a user who is both colour blind and an Australian, who should receive an image which is both grayscale and upside down, and also generalises the

image inversion condition for Antarcticans and Australians to derive from the fact that they both belong to the "Southern Hemisphere". This structure of contextual adaptations, with multiple sources of context all competing to advise the same target implementation which must remain "closed" in the Meyerian sense marks out this example as an instance of level 3 reuse in our taxonomy of section 4.

9.2 Reworking this example in Infusion

Our reworking of this example in Infusion[11] will differ in many ways from the original sample. We will demonstrate a fully working application for the web, which will show the image rendering process applied to a full framerate video stream, in order to highlight design issues which emerge from treating extremely high-performance use cases which cannot be met without the use of specific external languages and technologies — in this case, a "shader" written in the GLSL shader language of the HTML5 technology WebGL.

9.2.1 State transfer in the application

Following our discussion in section 7.2 on state transfer, we will organise our application's adaptation functions not around an individual function call `drawPixel` as in the original, but around the transfer of two matrices, one mapping the colour space of the renderer and one mapping the pixel space. There are strong practical as well as design incentives for this — if rendering a 1080p video at 60Hz, our `drawPixel` function would have to completely dispatch within 8ns even for a system fully saturating a core on this task, and we would naturally seek to defer to the GPU's own highly performant drawing loop rather than pushing it through an expensive dispatch process on the main core.

9.2.2 Factorising the example using context awareness

Infusion contains dedicated features for handling context-aware scenarios of this type, involving the use of the framework grade `fluid.contextAware` in the target component, allowing configuration to be built up in an options section named `contextAwareness` by means of some framework utilities (e.g. `fluid.contextAware.makeAdaptation`) and responding to context features marked to the current user by utilities such as `fluid.contextAware.makeChecks` and `fluid.contextAware.forgetChecks`. Full documentation is provided for these features at (ContextAwareness 2017). Note that all of these framework utilities represent "user-mode" convenience functions, which all factorise in practice down to Infusion primitive operations of the types seen in 6.3 — that is, registration of defaults holding options distributions, construction and destruction of components — they are thus equally well addressable over a remote HTTP

```
1
2   fluid.defaults("onward.imageRenderer", {
3       gradeNames: ["fluid.component", "fluid.contextAware", "fluid.createOnContextChange"],
4       mergePolicy: {
5           coordinateMatrix: fluid.arrayConcatPolicy,
6           colourMatrix: fluid.arrayConcatPolicy
7       },
8       coordinateMatrix: "@expand:onward.multiplyMatrices({that}.options.coordinateMatrix, \
9           onward.mat2ToMat3)",
10      colourMatrix: "@expand:onward.multiplyMatrices({that}.options.coordinateMatrix)"
11  });
12
13  onward.mat2toMat3 = function (mat2) {
14      return [mat2[0], mat2[1], 0,  mat2[2], mat2[3], 0,   0, 0, 1];
15  };
16
17  onward.multiplyMatrices = function (matrices, transformSpec) {
18      var transform = transformSpec ? fluid.getGlobalValue(transformSpec) : fluid.identity;
19      return fluid.accumulate(function (accum, extra) {
20          return mat3.multiply(accum, transform(extra));
21      }, mat3.identity(mat3.create()));
22  };
```

Listing 7. Matrix accumulation for image renderer

Nexus API as seen in listing 4 targeted at the runtime as well as over this local function call API.

On the JavaScript side we will maintain an "avatar" (Clark 2017) representing the state of the rendering process in a component of type `onward.imageRenderer` onto which we'll target our adaptations. Our adaptations will target the `imageRenderer` with various additional mixin grades which will contribute matrices into its lists when certain contexts match. Part of the definition of the renderer is shown in figure 7 — this encodes that it accepts two options named `coordinateMatrix` and `colourMatrix` which will simply be accumulated rather than merged during the options merging process. These give rise to two top-level options of the same names holding the accumulated matrix product of the constituent elements, by means of the expander definitions on lines 8-10. Expanders provide a means for Infusion configuration to be generated from function calls into the base language (JavaScript). These expanders are written in the "compact syntax" supported by Infusion as a syntactic sugar for the full JSON expander syntax. They are expanded to the JSON equivalents by a macro system before the system interprets them.

Let's first deal with the "southern Hemisphere" family of adaptations — a base context `o.c.southernHemisphere` encoding the common inversion context, `o.c.antarctic` which in addition encodes doubling of size, and `o.c.australia` which simply derives from `o.c.southernHemisphere`[12]. These adaptations appear in listing 8.

Line 27 shows an interesting example of arbitration between the priority of adaptations. This arbitration is unnecessary given the way we have encoded these adaptations, since all of the adaptation matrices commute — it doesn't matter which order they are composed in the renderer. Note that whilst our definitions are very significantly more verbose than the Korz equivalents, they are a little more cognitively economic since we don't need to reiterate both the scaling and inversion components of the mapping as in the Korz example of specialization for the Antarctic user which are shown in line 3 of listing 9.

---

[11] A live version of the application can be experimented with at `https://fluid-studios.github.io/infusion-onward-video-example`, and the full source code found in the source code held in the corresponding github repository

[12] First path segments `onward.contexts` abbreviated to aid line breaking

```
1   // Firstly, the context marker grades themselves
2   fluid.defaults("onward.contexts.southernHemisphere", {
3       gradeNames: "fluid.component"
4   });
5   fluid.defaults("onward.contexts.antarctic", {
6       gradeNames: "examples.contexts.southernHemisphere"
7   });
8   fluid.defaults("onward.contexts.australia", {
9       gradeNames: "onward.contexts.southernHemisphere"
10  });
11
12  // Secondly the adaptations binding occurence of contexts onto renderer adaptations
13  fluid.contextAware.makeAdaptation({
14      distributionName: "onward.adaptations.southernHemisphereInversion",
15      targetName: "onward.imageRenderer",
16      adaptationName: "yInversion",
17      checkName: "southernHemisphere",
18      record: {
19          contextValue: "{onward.contexts.southernHemisphere}",
20          gradeNames: "onward.rendererAdaptation.yInversion"
21      }
22  });
23  fluid.contextAware.makeAdaptation({
24      distributionName: "onward.adaptations.antarcticDoubling",
25      targetName: "onward.imageRenderer",
26      adaptationName: "scaleDoubling",
27      adaptationPriority: "before:yInversion", // no effect - just for priority demonstration
28      checkName: "antarctic",
29      record: {
30          contextValue: "{onward.contexts.antarctic}",
31          gradeNames: "onward.rendererAdaptation.doubleScale"
32      }
33  });
34
35  // Thirdly, the renderer adaptations
36  fluid.defaults("onward.rendererAdaptation.yInversion", {
37      coordinateMatrix: [[1, 0], [0, -1]]
38  });
39
40  fluid.defaults("onward.rendererAdaptation.doubleScale", {
41      coordinateMatrix: [[2, 0], [0, 2]]
42  });
43
```

Listing 8. Adaptations dealing with inversion and scaling for southern hemisphere

```
1   method { rcvr ≤ screenParent , location ≤ antarctica }
2   drawPixel(x, y, c) {
3       {-location}.drawPixel(2 * x, -2 * y, c);
4   }
```

Listing 9. Korz slot definition for Antarctic user

```
1   fluid.contextAware.makeAdaptation({
2       distributionName: "onward.adaptations.colourBlindMonochrome",
3       targetName: "onward.imageRenderer",
4       adaptationName: "colourAdaptation",
5       checkName: "colourBlind",
6       record: {
7           contextValue: "{onward.contexts.colourBlind}",
8           gradeNames: "onward.rendererAdaptation.monochrone"
9       }
10  });
11
12  // Standard luma coefficients from https://en.wikipedia.org/wiki/Luma_(video)
13  fluid.registerNamespace("onward.constants");
14  onward.constants.lumaRow = [0.2126, 0.7152, 0.0722];
15
16  fluid.defaults("onward.rendererAdaptation.monochrome", {
17      colourMatrix: [
18          onward.constants.lumaRow,
19          onward.constants.lumaRow,
20          onward.constants.lumaRow
21      ]
22  });
```

Listing 10. Colourblind/monochrome adaptations

The listing 10 shows the adaptations for the user who is colour blind.

9.2.3 Commentary on the sample

Our rendition of this sample is very substantially more verbose than that of the Korz treatment. Firstly, this is a result of the fact that we have no syntax. Encoding of all Infusion primitives as JSON naturally increases their verbosity greatly. However, this directly serves several of our ends: firstly, the ecosystem of tools consuming and producing these artefacts is hugely simplified.

Secondly, much of our verbosity stems from the seemingly redundant extra names which our definitions are decorated with. For example, each adaptation block includes not only a `distributionName` but also an `adaptationName` and `checkName`. These additional names directly serve the aims of the OAP, in ensuring that the authorial landscape is left with sufficient landmarks for further authors to bind on to, at each level of granularity. The `adaptationName` can be targeted with a priority-based expression by further adaptations, indicating which adaptation should defer to which other, and the `distributionName` can be used to identify the distribution itself in a running system in order for it to be overridden. Existing treatments of Korz, for example, don't specify the following scenarios:

- The disposition when multiple definitions are received with apparently identical guard conditions — does a later definition displace an earlier, and how are ambiguities resolved with respect to minor mismatches in condition wording or aliases in term names?

- How the author might arrange for the situation where a slot specification with fewer guard conditions is required to take priority over one with a greater number (the opposite of the semantic built in to the language's design)

Giving each such definition in the system a unique name is essential to allowing these contextually-guarded definitions themselves a first-class authorial status and ensuring that our algebra of expressions remains closed.

Another substantial difference between Korz and Infusion is that, whilst Infusion is *symmetric* (in terms of establishing symmetry between multiple authors allocating context dimensions), it is not *subjective*. Each observer of a particular assemblage of components will see exactly the same state and behaviour. This seems essential to us in order to make it clear how a particular part of the system should be externalized in order to represent it in another system. (Ungar 2014) doesn't fully specify how contextual information should be attached to the observer and tracked, suggesting that it might be associated with the call stack. We consider this might be hard to manage when the interaction is part of one or many long-running asynchronous processes involving interactions stemming from multiple users. We suggest that a structural scoping model, where particular areas of the component tree are stably associated with interaction on behalf of different users in different contexts, would lead to clearer, more performant and more easily externalised designs.

We should observe that what passes between the portions of the systems written in different languages (Infusion/JavaScript and GLSL) is nothing specially bound to a particular language system or Infusion specifically — it is a collection of matrices in a straightforward encoding. This situation, where Infusion ends up as the gatekeeper for material representing some publicly encoded *lenses* rather than simply a broker for message passing, is quite typical, and

```
1   {
2     type: "untrusted.development.all.local",
3     options: {
4       gradeNames: "kettle.multiConfig.config",
5   ...
6         distributeOptions: {
7           "untrusted.development.port": {
8             record: 8088,
9             target: "{that cloudBasedConfig}.options.mainServerPort"
10          },
11          "untrusted.development.prefs": {
12            record: "http://localhost:8088/preferences/%userToken",
13            target: "{that cloudBasedConfig flowManager preferencesDataSource}.options.url",
14            priority: "after:flowManager.development.prefs"
15          },
16    ...
17      }
18  }
```

Listing 11. GPII FlowManager configuration showing level 4 reuse

such lenses form frequently encountered material in Infusion designs — see (Basman 2015).

Finally, we should observe that whilst our example is capable of processing high-resolution video at framerate on everyday equipment, the space of adaptations of our version of the sample is much smaller than the Korz original. We are only able to make adaptations which are *linear* in the colour and coordinate spaces, whereas the Korz example can perform arbitrary functional mappings.

10.   A Real-World Example of Type 4 Reuse

The Global Public Inclusive Infrastructure (GPII - (Vanderheiden 2011)) is an ambitious project whose aim is to implement an auto-personalisation system makingt the resources of operating system and application-level adaptation available to users across all applications and platforms.

A core GPII architectural component is the `flowManager` (FlowManager 2017) which assembles the user's preferences, the capabilities of a local device and relevant privacy policies, and orchestrates the device's capabilities to bring it to an inferred condition meeting the preferences.

We analyse the network of authors who collaborated on the design of `flowManagers` in terms of the OAP. The initial design by author A implemented a "local `flowManager`" which was simply hosted on the user's machine. Author B extended A's design to incorporate a remote "cloudBased `flowManager`" to factor off just those functions which were relevant within the cloud. Author C then aggregated together the designs of authors A and B to create a composite "untrusted `flowManager`" in order to meet freshly characterised privacy concerns, and needed to direct overriding configuration simply at B's portion of the design, whilst leaving A's untouched. We should 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 listing 11 we see a section of author C's configuration.

On line 12, an IoCSS selector with 4 components {that cloudBasedConfig flowManager preferencesDataSource} selects by path a particular subcomponent of just the remote `flowManager` component for "advice". This full specification is neces-

sary 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 a traditionally constructed system, the "substitutability" of these two instances via inheritance the `gpii.flowManager` base grade would require work proportional to the size of the total design in order to resolve. Using this IoCSS selector, one author is able to compactly specify a "diff" (see section 8.1.1) with respect to the design of another, whilst leaving the original design unmodified. Note that, as in our context-aware sample of section 9.1, these authors are obliged to continually issue fresh names to keep pace with their rate of creating fresh surfaces for adaptation.

As a result of this, the design now stands ready for affordable 4th and 5th-order reuse: A further author, a mid-level integrator D of the GPII, will aggregate a distribution from components representing a particular set of use cases, and a yet further author E will derive a further distribution from this aggregation, whilst specifying that a deeply nested (and without a means such as IoCSS, hard to name) component should have a specially configured value. This is done whilst still leaving the field relatively uncluttered for yet further integrators to appear, and in their turn make further configurations, possibly overriding yet again the choices of E.

We have a truly open authorial graph.

11.   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 argue represent everyday levels of reuse that arise routinely in real architectures. We've argued that the key to eliminating horizons in the graph of authors is to allow program differences to be freely expressed and combined as programs. This expression gives rise to the fundamentally different architectural strategies informing the design of Fluid's Infusion system. These strategies also promote more natural externalisation of designs, supporting more straightforward interactions with external systems implemented in different processes, languages and idioms.

We have exhibited the Open Authorial Principle, which summarises the requirements of all 4 levels of this tower as well as encompassing a much wider terrain of as yet unarticulated reuse capabilities. We explain that meeting this principle well 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.

References

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford and P. Tucker Withington *A monotonic superclass linearization for Dylan*, Proceedings of the 11th ACM OOPSLA, pages 69-82, ACM, New York, 1996

Antranig Basman, Colin Clark and Clayton Lewis *Harmonious Authorship from Different Representations*, Proceedings of the 26th Annual PPIG Workshop, 2015

Antranig Basman, Clayton Lewis, and Colin Clark *To inclusive design through contextually extended IoC*, Proceedings of the ACM OOPSLA Companion (Wavefront), 2011.

Gilad Bracha *A DOMain of Shadows*, blog posting at `http://gbracha.blogspot.co.uk/2014/09/a-domain-of-shadows.html`

Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox and Eliot Miranda *Modules as Objects in Newspeak*. Proceedings of the 24th ECOOP, June 21-25 2010. Springer Verlag LNCS 2010.

Colin Clark and Antranig Basman *Tracing a Paradigm for Externalization: Avatars and the GPII Nexus*, ⟨Programming⟩, Proceedings of Salon des Refusés Workshop, 2017.

Pascal Costanza and Robert Hirschfeld *Language Constructs for Context-oriented Programming: an Overview of ContextL*, in: DLS'05: Proceedings of the 2005 Symposium on Dynamic Languages, ACM, New York, NY, USA, pages 110, 2005.

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

Roy T. Fielding *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, University of California, Irvine, 2000.

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

Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

GPII Team *The GPII Nexus*, `https://wiki.gpii.net/w/The_Nexus`, 2017.

GPII Team *The GPII FlowManager*, `https://wiki.gpii.net/w/Flow_Manager`, 2017

Fluid Team *Fluid Infusion Documentation*, `http://docs.fluidproject.org/infusion/development/`, 2017.

Fluid Infusion Documentation on ContextAwareness feature `http://docs.fluidproject.org/infusion/development/ContextAwareness.html`, 2017.

Alan Kay *"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`

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin *Aspect-oriented programming*. Proceedings of the 11th ECOOP (1997).

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

Stephen Kell, Danilo Ansaloni, Walter Binder and Lukáš Marek *The JVM is not observable enough (and what to do about it)*, Proceedings of the VMIL '12, pages 33-38, ACM, New York.

Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay and Michel Beaudouin-Lafon *Webstrates: Shareable Dynamic Media*, Proceedings of the 2015 UIST, pages 280-290, ACM, New York.

Robert C. Martin *The Open-Closed Principle*, C++ Report, January 1996

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

David L. Parnas *On the Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM 15 (12) pp 1053-58, ACM, New York, December 1972

David Ungar, Harold Ossher and Doug Kimelman *Korz: Simple, Symmetric, Subjective, Context-Oriented Programming*, Proceedings of the Fourth Symposium on New Ideas in Programming and Reflections on Software (Onward), ACM, 2014

Gregg Vanderheiden and Jutta Treviranus *Creating a Global Public Inclusive Infrastructure*. Universal Access in Human-Computer Interaction — Design for All and eInclusion, pages 517-526. Berlin: Springer, 2011.

W3C *XML Path Language (XPath) 3.1* W3C Recommentation 21 March 2017.