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 through aspect-oriented, subject-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 consider the implications of our principle for the externalisation of application designs, resulting from the need to promote the representation of differences between programs as valid programs themselves.

Keywords   context awareness, declarative configuration

0.   The Open Authorial Principle

The design should allow the effect of any expression by one author to be replaced by an additional expression by a further author.

We propose that supporting this principle is so important that we should uproot many of our ideas about how good software is built, along with most of our common tools, technologies and means of technical expression. This paper identifies a historical axis of development towards increasingly generous modes of reuse, but argues that much more radical progress is possible and indeed desirable. We imagine uprooting information hiding, function composition and scopes, the program stack, compilers and programming languages in general. Progress towards this revolution will be necessarily incremental and involve many losses along the way — in this paper we describe a few steps we have taken and sketch out a wider trajectory over terrain whose structure we only expect to become clearer once we approach it.

Our justification for the principle will be a mixture of social, economic and technical concerns which cannot always be cleanly disentangled. Primarily we will argue on the basis of the role we desire software to have in society, rather than from the usual technological or mathematical considerations of correctness, consistency or efficiency. Along the way we will note some of the philosophical, economic and organisational background that has led to the kinds of software that we currently have, and why we consider these justifications are not applicable.

We will revisit the principle from various points of view throughout the paper, and progressively unpack some of its implications. We begin by considering an activity which the principle facilitates, *reuse*.

1.   Introduction

Reuse is the capacity of a design to empower others to continue the design process via extension or adaption. We will look at reuse by following the histories of design artefacts as they pass between the hands of authors whom we see as comprising networks. These histories form what we will call "authorial stories" involving authors conventionally labelled A, B, C, etc. in which collection an "end user" E is morally incorporated. The design artefact is composed of what we call the *expressions* of the authors, by which we mean whatever they write or whatever gestures they make to convey their design intentions. The resulting design has an *effect*, by which we mean the design as experienced by someone in the role of use (e.g. E). 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.

Some developments in programming idioms have increasingly supported reuse as supported by our principle, by supporting reuse 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[1] according to the sophistication of the reuse stories that they support, starting

---

[1] Note that levels 3 and 4 are not strictly nested, but in practice addressing level 4 scenarios seems to entail dealing with those at level 3.

with object-orientation at the base level 1, and ending with a presentation of our own system, *Infusion*, whose development was motivated by the more complex reuse stories that we see at level 4 and beyond.

1.1 Horizons in the network of authors

We consider that all existing programming idioms create unwelcome distinctions among a population of authors, and so create *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. We discuss the scaling economics of development and of communication amongst authors in section 5.6.

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 Dependency Injection (Fowler 2004), and at level 3 more modern and ambitious systems such as Newspeak (Bracha 2010) and Korz (Ungar 2014). We will situate this hierarchy of levels, showing the way to level 4 and beyond, under our *open authorial principle* stated in section 0, which we will elaborate in section 5. Supporting the principle has driven the design of our system, Infusion, which we describe in section 6.

1.2 An algebra of program differences

The Open Authorial 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 section 5.3 we will informally consider a *program addition operator* ⊕ combining members of the algebra. This operator lies outside the space traditionally considered part of a programming language design.

1.3 Reuse levels

The following sections will tour our hierarchy of reuse levels, an overview and illustrations of which appear in table 3. 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) states that

> *A module should be available for extension (*open*) but also available for use (*closed*),*

where, for Meyer, "available for use" meant that a module's content should not be modifiable, promoting uses such as caching, verification, etc.. Meyer's principle 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.

2.1 Toy example — a flowManager

We will here begin a running example of reuse that we will follow through levels 1, 2 and 4 of our reuse hierarchy (level 3 will be treated with a separate example in section 8.1). We will consider reuse stories centering around a base artefact called a "flowManager", a toy example simplified from our implementation of the GPII's auto-personalisation system[2].

In the abstract, it is a "class" flowManager holding a method, keyIn which accepts some credentials from the user, and brings the system to the appropriate state. We'll ignore all the arguments and effects of this method and almost all other details since we are concentrating on the schematics of reuse.

In Meyer's presentation[3], there are implicitly authors A, B, and (at least one) end user E. Let's consider that A has written a flowManager capable of running on a local device, and B has provided a refinement of it known as an "untrusted flowManager" which provides the same service for the user, but does so by following a more stringent security policy. Listing 1 shows this situation encoded in our sys-

---

[2] The Global Public Inclusive Infrastructure (GPII - (Vanderheiden 2011)) is an ambitious project whose aim is to implement an auto-personalisation system providing operating system and application-level adaptation to users across all applications and platforms. The GPII's flowManager (FlowManager 2017) 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.

[3] 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.

```
1   // Author A, artefact alpha
2   fluid.defaults("gpii.flowManager.local", {
3       gradeNames: "gpii.flowManager",
4       invokers: {
5           keyIn: {
6               ....
7           }
8       }
9   });
10
11  // Author B, artefact alpha'
12  fluid.defaults("gpii.flowManager.untrusted", {
13      gradeNames: "gpii.flowManager.local",
14      invokers: {
15          keyIn: {
16              ....
17          }
18      }
19  });
20
21  // Author E consumes alpha or alpha'
22  fluid.defaults("gpii.flowManagerTester", {
23      gradeNames: "fluid.component",
24      components: {
25          underTest: {
26              type: "gpii.flowManager.local"
27          }
28      }
29  });
30  // Author F executes the test
31  gpii.flowManagerTester();
```

Listing 1. A small Infusion example showing basic Meyerian derivation

```
1   // Author A, artefact alpha
2   class LocalFlowManager extends FlowManager {
3       keyIn() {
4           ....
5       }
6   }
7
8   // Author B, artefact alpha'
9   class UntrustedFlowManager extends LocalFlowManager {
10      keyIn() {
11          ....
12      }
13  }
14
15  // Author E consumes alpha or alpha'
16  var FlowManagerTester = function(underTest) {
17      ...
18  }
19  // Author F executes test
20  FlowManagerTester(new LocalFlowManager());
```

Listing 2. The Listing 1 example rendered in ECMAScript 6

tem, Infusion — whilst a fuller introduction to Infusion will wait until section 6, our usage here should be simple enough to be transparent, and for comparison we have encoded the same relationship in a more familiar Meyerian system, ECMAScript 6 classes, in Listing 2. In this case our end user E is a tester wishing to verify functions of the flowManager. In the last line of each sample, we split E's role further to assign to a further author, F, the task of actually executing the test function, anticipating future discussion.

From the point of view of E, an untrusted flowManager is a substitutable replacement for a local flowManager. This substitutability may or may not be encoded in some form of *base class* or *interface* here named a simple "flow manager" (`gpii.flowManager/FlowManager`) and deliberately left out of scope. A vast literature on the nature of this substi-

tutability and how it may be encoded in program text stems from (Liskov 1988), and a different approach to this issue has resulted in (Martin 1996)'s re-presentation of Meyer's principle, but this is a detail that is uninteresting to us here. Any of the reasonable choices of approach available in this area may be made without impacting the current discussion.

For the purposes of an abstract presentation, we will give symbolic labels to the design elements which so far have been concrete. A's original unit, the local flowManager, is α, B's refinement to the untrusted flowManager α'. Any base contract on substitutability which the language permits (`gpii.flowManager`) is named $\aleph$ (aleph), which may also have a dual role as a base implementation artefact. This situation is illustrated in row 1 of table 3.

Note that Meyerian inheritance can already land us with reuse problems even in this simple situation. Even if α' is indeed an effective substitute for E's use of α (whether or not this substitutability can be encoded successfully in $\aleph$), the substitutability can only be enacted if E's only use is to *accept* an α rather than needing to construct one. If E's code creates an α, it can't instead create an α' without being modified. Author F, the one who actually executes the test function, suffers from exactly this problem, since it is they who must be responsible for constructing the object under test and hence designating its type. 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 3.2. In Infusion we can shift the burden of adaptation either from F to E or indeed any further author in the system by techniques we will present in section 6.

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. Let us say that Author A's local flowManager has a subunit that we name a "preference data source". This represents some form of access to persistence where the user's preferences are stored. Whilst different implementations of this data source may conform to the same ostensible contract in terms of data provision, they will differ amongst themselves with respect to some kind of cross-cutting concern, for example whether the preferences are stored locally or remotely, or whether the preferences have been filtered with respect to the user's privacy policy. Let's say that author A's implementation is a "local data source" `gpii.dataSource.local` , and that author B's refinement is of this is to an "untrusted data source " `gpii.dataSource.untrusted`. Listing 4 shows the Infusion representation of this reuse, where B has been able to provide their refined implementation to E without requir-

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, DI, 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 3. Table of levels of increasingly sophisticated reuse with illustrations

ing A to change their definition, and Listing 5 shows the rendering of this situation as ECMAScript classes.

In Listing 5, author B has incurred further costs since in addition to implementing the untrusted data source, they have also needed to create a variant flow manager to contain it. This is the beginning of the increased design scaling costs we alluded to in section 1.1, which become more severe with increasing reuse level for languages that are not adapted to them. Corresponding to the abstract presentation in row 2 of table 3, the local flow manager is α, the local data source is β, the untrusted data source is β' and the adapted flow manager is α'.

3.1 Mitigating scaling costs

We have been a little unfair to Meyerian inheritance here, since by an adjustment to the design we could to a fair extent mitigate the scaling costs. By shifting responsibility for the value of `preferencesDataSource` into the `LocalFlowManager`'s constructor arguments, we could arrange to retain its implementation. This however involves us in two issues:

The need for foresight The LocalFlowManager implementor may simply not have considered that parameterisation of this aspect of their implementation would be useful or permissible to their users. Without the foresight of explic-

itly cascading this DataSource dependency, the improved design scaling is not available to downstream authors.

Scaling of constructors As components scale up to be the head of an increasingly large tree of nested dependencies, the number of arguments to be recursively surfaced in each constructor grows exponentially with the depth of the tree. This represents a design scaling cost in its own right.

3.2 Factories, Dependency Injection and Newspeak

There are numerous solutions in the literature to this relatively mundane reuse situation. Some apply a "factory pattern" — instead of supplying ready-built objects as constructor arguments, we instead supply functions which dispense them, which may themselves be polymorphic methods on substitutable objects. This leads to further scaling issues since we have a fresh class of entity — factories — 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 reuse 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-

```
1   // Author A, artefact alpha
2   fluid.defaults("gpii.flowManager.local", {
3       gradeNames: "gpii.flowManager",
4       components: {
5           preferencesDataSource: { // member beta
6               type: "gpii.dataSource.local"
7           }
8       }
9   });
10
11  // Author B, creates a variant beta' of beta
12  fluid.defaults("gpii.dataSource.untrusted", {
13      gradeNames: "fluid.component"
14      ...
15  });
16
17  // Author E can consume alpha plus beta' without modification
18  fluid.defaults("gpii.flowManagerTester", {
19      gradeNames: "fluid.component",
20      components: {
21          underTest: {
22              type: "gpii.flowManager.local",
23              options: {
24                  components: {
25                      preferencesDataSource: {
26                          type: "gpii.dataSource.untrusted"
27                      }
28                  }
29              }
30          }
31      }
32  });
33  // Author F executes the test
34  gpii.flowManagerTester();
```

Listing 4. A small Infusion example showing reuse under parameterised composition

```
1   // Author A, artefact alpha
2   class LocalFlowManager extends FlowManager {
3       constructor: {
4           this.preferencesDataSource = new LocalDataSource();
5       }
6   }
7
8   // Author B, artefact beta'
9   class UntrustedDataSource;
10
11  // Author B now must also write artefact alpha'
12  class UntrustedFlowManager extends LocalFlowManager {
13      constructor: {
14          this.preferencesDataSource = new UntrustedDataSource();
15      }
16  }
17
18  // Author E consumes alpha or alpha'
19  var FlowManagerTester = function(underTest) {
20      ...
21  }
22  // Author F executes test - must also select alpha or alpha'
23  FlowManagerTester(new UntrustedFlowManager());
```

Listing 5. The Listing 4 example rendered in ECMAScript 6

class design status. 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 3.3. This category of reuse problem also gave rise to a large family of frameworks and techniques based on "dependency injection" (DI) paradigmatically described in (Fowler 2004).

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 ev-

ery point of consuming a type name, a user has the facility to rebind it to one resolvable in their context. Indeed, (Bracha 2010) explicitly states that Newspeak "eliminates the primary motivation for dependency injection frameworks". As a result, Newspeak resolves not only reuse problems at this level but also some at the following level 3. However, as a result of its "hermetic axiom", that "a modules only connection to the outside world comes from the actual arguments passed to the factory method that created it", it may not directly solve reuse problems at level 4 and beyond because of the impossibility of structural inspection of a wider system from within itself. Also, Newspeak fails to tackle "deep hierarchy" level 3 reuse problems in a scalable way because overriding of nested classes needs to proceed through one step of the hierarchy at a time.

3.3 Containment through inheritance

What if A had already been applying Meyerian reuse and the containment-like relation between α and β was already inheritance itself? With a simple use of implementation inheritance without overriding, we might say that α "IS-A" β through including β's entire definition into its own. B's reuse requirement is now expressed as wanting an α' which is α with its base class β replaced by β'. Unfortunately, this is an impossible form of reuse via the inheritance relation designed into traditional OO languages — whilst one can override elements of one's base class, one cannot override its actual specification. This 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.

3.3.1 Reuse through parameterised types

In C++, author A, perhaps trying to address the reuse situation of section 3.3, 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. The requirement for this pattern of reuse was encountered very early in the lifetime of the C++ language and became characterised as the "curiously recurring template pattern"(Coplien 1995).

Parameterised types are a sufficiently powerful reuse mechanism that they also resolve the aggregation variant

of this problem in section 3.2 — it's just as easy for a parameterised type to appear as the type of a member as the type of a base class.

3.3.2 Containment through private use

The point within α where β is used may also lie within arbitrary implementation code, rather than a member initialiser appearing in a constructor, and hence 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. This form of "private reuse" occurs, for example, whenever two functions are composed using traditional programming language mechanisms, leading to our assertion in the opening paragraph that we would like to see function composition uprooted. Consult (Basman 2017) for more details of this argument.

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 OO languages such as Java and C++. With AOP, author B is allowed to create a symbolic expression known as a *joinpoint* to name the point in A's design where β is referred to. A further expression known as *advice* encodes the modification of the design where β is substituted by β'.

Whilst AOP provides a clear native solution to the level 2 reuse problems presented earlier in this section, it fails with the more demanding level 3 and 4 scenarios we will present in section 4. The key limitation of AOP in these 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 requires something new. That is, in the terminology where we elaborate the Open Authorial 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[4], these also suffer from the same problem noted against templates in section 3.3.1 — 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, DI and similar formalisms, only represents level 2 reuse. In practice, much more demanding scenarios arise quite regularly. For example

---

[4] 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 chain of decorators attached to the same base artefact.

- Level 3 reuse scenarios involve several authors, B, C, etc. who have written modifications to modify the same part (e.g. α) of A's work. This is not the situation addressed through "multiple inheritance" or "mixins' provided in some flavours of OO since we require A's original expression to be consumed unmodified by E, without any further authors who simultaneously want to put B's, C's expressions in scope needing access to a construction point of α or suffering a scaling burden through having to refer to each other's expressions.

- 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 3, 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 (copied into our appendix B.2), 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 3, 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

In this section we expand on our original statement of the principle in section 0 by recognising it as a generalisation of Meyer's open/closed principle, and making a restatement of the principle by considering its implication for author expressions forming an algebra of program differences.

5.1 Reappraising Meyer's Principle

Meyer's open/closed principle is a good foundation for ours. We believe 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 constant except in the face of genuine revisions to the overall program. 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 point of authorial control expressing 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*.

5.2 Alternative Statement of the 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 0:

The design should allow the effect of any expression by one author to be replaced by an additional expression by a further author.

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

The design should be drawn from a closed algebra of expressions which will enable an open graph of authors.

5.3 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 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 to 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 to be fetched and interpreted. To be functionally open, the system's facility for addressing $\oplus$ must 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.

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.

5.4 Distinction to previous program algebras

Our algebra should not be confused with a similarly-named structure which has emerged many times in the literature, for example in (Backus 1978)'s "algebra of programs". Rather than imagining an algebra whose combining operation is merely the symbolic combination of mathematical expressions representing the program fragments, our combining operation represents *whatever combining operation is necessary in the world of the actually executing program* in which expressions are combined. Furthermore, we are not so much interested in the ability simply to build up complex programs from simpler ones, which is a facility which emerges in practically every programming language. Instead, we are more interested in the practical capability, presented with two already written or planned programs, to decompose the difference between them as a reasonably plausible program expression in its own right. This is expressed by our defining property in section 5.3 being expressed in a decompositional rather than an additive form.

5.5 The principle cannot be provably or fully satisfied

Conformance to the principle 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 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.

Another important reason that the principle cannot be fully satisfied is that increasing attempts to satisfy it inevitably entail losses in other design areas, which we will discuss in the next section.

5.6 Economics of development and communication

The OAP is stated in isolation as an apparently absolute principle, but naturally it is just one element in a wider picture of design economics. In practice, increasing attempts to satisfy the OAP will result in designs with weaker locality of reference, reduced comprehensibility of individual design artefacts (in the absence of supporting tools), greater verbosity, and greater consumption of runtime resources. Linked goals of this paper are:

  i) To argue that previously unvisited extremes of the design tradeoffs implied by satisfying the OAP should be explored by designing new language-like systems

 ii) To exhibit what the design costs incurred by these systems look like

iii) To suggest ways that these costs could be reduced either through improved engineering of the supporting system or improved tooling

An important scaling phenomenon that is directly implied by the economics of the OAP is one described by (Brooks 1995), that the costs of communication between a team of n authors grows superlinearly with n — and therefore, one of the goals of this paper, to bring the growth in costs for designs worked by multiple authors to close to linear might appear inherently futile. However, we argue that Brooks' analysis is inapplicable to the situations encountered by many communities of authors today. Brooks' analysis was situated within the "military-industrial complex" model of organisation where hierarchically organised groups are assembled in more or less the same time and place in order to achieve a task which results in clearly quantifiable economic benefits. The experience of the current authors with software development has frequently violated these assumptions:

- Where one's "collaborators" may not only be widely dispersed but in many cases unavailable — they have left an major open source project drifting and unmaintained, or are trying to achieve widely different objectives than your own

- Where one is attempting to build a "product" whose function can't be easily characterised, and tends to drift over time

- Where one can't directly quantify the benefits of the software since it is not the artefact which is the agency of the value, and further, one may not easily draw a boundary around its community of use

On these grounds, Brooks' observation that the number of communication links grows superlinearly between programmers in a "team" of size n becomes questionable since in many cases the links are either unidirectional or else completely absent. One of the main aims of the OAP is to make software development tractable in these "open graphs" of authors where one must frequently inherit design artefacts from others with little or no influence over or communication with their authors — and still worse, when this lack of influence has been cascaded down a long chain of project dependencies.

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 since 2008. Many of Infusion's features were designed to support the OAP, but it also aims to meet many other needs (dataflow programming, live programming — see (Basman 2016), 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 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[5].

---

[5] 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 towards helping users to find relevant objects at runtime and adapt

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 and ourselves with the different behaviour of the related constructs in OO.

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 A.1.

6.2.1 References and Distributions

A component's merged options reflect more than just the values expressed in the sources. 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 authorial 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, data-driven process, and only rejects graphs which are cyclic with respect to individual leaf values.

6.3 A minimal Infusion program

Listing 1 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** implemented as part of the GPII's Prosperity4All Program (Nexus 2017), all of the facilities used in listing 1 could be addressed from outside the process using standard HTTP

---

them, preferably before their immediate clients start to consume them – see section 3 for examples. In the experience of the authors, the economics of adaptation for ordinary dynamic languages usually reduce to those for static languages — locating the source code for the artefact, forking and modifying it.

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

Listing 1. A minimal Infusion program

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

Listing 2. Listing 1 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 ",
22     that.monty.options.siblingAge);
```

Listing 3. A small Infusion example showing reference resolution

endpoints. For example, the sequence of HTTP requests in listing 2 would have the same effect as in listing 1.

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 3 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 3 is known as an IoC reference, named after one of Infusion's earlier roles 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

Following section 1.2, we now discuss two closely related properties of Infusion's design which we argue must emerge in a design or system that strongly satisfies the OAP.

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, for example by promoting the expression of avatars (Clark 2017).

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 role in structuring an application, as 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 for 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 A.1.

7.2 Externalisability and REST

A systematic failure of object-oriented environments is their tendency to be "hermetic", that is, to give insufficient consideration to what lies outside the system. 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 nor capacious. In section A.2.1 we show an example of how an externalisable state idiom can eliminate a message passing linkage that would require untenable throughput.

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, and Clark (2017) notes that the choices of some MIDI devices to respond to certain messages by simply dumping some memory contents has greatly improved to their longevity and adaptability.

---

[5] for further details, see (Basman 2016) for "Queen of Sheba adaptation".

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 out 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 the distribution is to be targeted[6]. 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 4 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 `colourizer` component defined at line 13, 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 targetting 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 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

---

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

Listing 4. Options distribution in level 3 reuse Infusion sample

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

We have reworked 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 of 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.

Despite this similarity in fundamental argument, the resulting architectures of Korz and Infusion are extremely different. To start with, Korz (like Newspeak) can function as

```
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 5. GPII FlowManager configuration showing level 4 reuse

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, 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.

The working out of the example is bulky and so has been pushed out to appendix A.

10. A Real-World Example of Type 4 Reuse

We analyse the network of authors who collaborated on the design of `flowManager`s 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 5 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 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 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 A.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. Meeting this principle well requires iterated cycles of implementation, self-observation and refinement, steadily increasing the range of expressions which the principle can subsume in a realistic space of user designs. We devote ourselves to this struggle.

A. Freely Dimensioned Context Awareness Sample

In this appendix section we present a worked example showing a level 3 reuse story.

A.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.

A.2   Reworking this example in Infusion

Our reworking of this example in Infusion[7] 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.

A.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.

A.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 Nexus API as seen in listing 2 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 listing 6 — 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

---

[7] 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.

```
fluid.defaults("onward.imageRenderer", {
    gradeNames: ["fluid.component", "fluid.contextAware", "fluid.createOnContextChange"],
    mergePolicy: {
        coordinateMatrix: fluid.arrayConcatPolicy,
        colourMatrix: fluid.arrayConcatPolicy
    },
    coordinateMatrix: "@expand:onward.multiplyMatrices({that}.options.coordinateMatrix,
        onward.mat2ToMat3)",
    colourMatrix: "@expand:onward.multiplyMatrices({that}.options.colourMatrix)"
});

onward.mat2toMat3 = function (mat2) {
    return [mat2[0], mat2[1], 0,  mat2[2], mat2[3], 0,   0, 0, 1];
};

onward.multiplyMatrices = function (matrices, transformSpec) {
    var transform = transformSpec ? fluid.getGlobalValue(transformSpec) : fluid.identity;
    return fluid.accumulate(function (accum, extra) {
        return mat3.multiply(accum, transform(extra));
    }, mat3.identity(mat3.create()));
};
```

Listing 6. Matrix accumulation for image renderer

product of the constituent elements, by means of the expander definitions on lines 8-10[8].

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`[9]. These adaptations appear in listing 7.

Line 27 shows how to arbitrate priority between adaptations, when needed. It's not actually needed here given the way we have encoded these adaptations, since all of the adaptation matrices commute. Note that whilst our definitions are 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 is shown in line 3 of listing 8. The complete listing of (Ungar 2014)'s Korz code for this sample appears in appendix B.1.

The listing 9 shows the adaptations for the user who is colour blind.

A.2.3 Commentary on the sample

Our rendition of this sample is substantially more verbose than the Korz treatment. Firstly, this results from Infusion's lack of syntax. Encoding Infusion primitives as JSON increases their verbosity greatly. However, this directly serves several of our ends: firstly, the ecosystem of tools consuming and producing these artefacts is much simplified.

Secondly, much of our verbosity stems from the extra names which our definitions are decorated with. For example, each adaptation block includes not only a `distributionName`

---

[8] 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.

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

```
// Firstly, the context marker grades themselves
fluid.defaults("onward.contexts.southernHemisphere", {
    gradeName": "fluid.component"
});
fluid.defaults("onward.contexts.antarctic", {
    gradeNames: "examples.contexts.southernHemisphere"
});
fluid.defaults("onward.contexts.australia", {
    gradeNames: "onward.contexts.southernHemisphere"
});

// Secondly the adaptations binding occurence of contexts onto renderer adaptation
fluid.contextAware.makeAdaptation({
    distributionName: "onward.adaptations.southernHemisphereInversion",
    targetName: "onward.imageRenderer",
    adaptationName: "yInversion",
    checkName: "southernHemisphere",
    record: {
        contextValue: "{onward.contexts.southernHemisphere}",
        gradeNames: "onward.rendererAdaptation.yInversion"
    }
});
fluid.contextAware.makeAdaptation({
    distributionName: "onward.adaptations.antarcticDoubling",
    targetName: "onward.imageRenderer",
    adaptationName: "scaleDoubling",
    adaptationPriority: "before:yInversion", // no effect - just for priority de
    checkName: "antarctic",
    record: {
        contextValue: "{onward.contexts.antarctic}",
        gradeNames: "onward.rendererAdaptation.doubleScale"
    }
});

// Thirdly, the renderer adaptations
fluid.defaults("onward.rendererAdaptation.yInversion", {
    coordinateMatrix: [[1, 0], [0, -1]]
});

fluid.defaults("onward.rendererAdaptation.doubleScale", {
    coordinateMatrix: [[2, 0], [0, 2]]
});
```

Listing 7. Adaptations dealing with inversion and scaling for southern hemisphere

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

Listing 8. Korz slot definition for Antarctic user

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?

```
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.monochrome"
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 9. Colourblind/monochrome adaptations

- 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 system 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 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

```
1   def {} pointParent = newCoord;
2   def {} point = newCoord extending pointParent;
3
4   var {rcvr  point} x;
5   var {rcvr  point} y;
6   var {rcvr  point} color;
7
8   method {
9       rcvr  pointParent,
10      device //dimension required but can be anything
11  }
12  display {
13      device.drawPixel(x, y, color)
14  };
15
16  def {} screenParent = newCoord;
17  def {} screen = newCoord extending screenParent;
18  method {rcvr  screenParent} drawPixel(x, y, color) {
19  // draw the pixel in the color
20  }
21
22  def {} locationParent = newCoord;
23  def {} location = newCoord extending locationParent;
24  def {} southernHemi = newCoord extending location;
25  def {} australia = newCoord extending southernHemi;
26  def {} antarctica = newCoord extending southernHemi;
27
28
29  method { rcvr  screenParent, isColorblind  true }
30  drawPixel(x, y, c) {
31      {isColorblind: false}
32          .drawPixel(x, y, c.mapToGrayScale)
33  }
34
35  method { rcvr  screenParent, location  southernHemi }
36  drawPixel(x, y, c) {
37      { -location }.drawPixel(x, -y, c)
38  }
39
40  method {
41      rcvr  screenParent,
42      isColorblind  true,
43      location  southernHemi
44  }
45  drawPixel(x, y, c) {
46      {-isColorblind}.drawPixel(x, y, c.mapToGrayScale);
47  }
48
49  method { rcvr  screenParent, location  antarctica }
50  drawPixel(x, y, c) {
51      {-location}.drawPixel(2 * x, -2 * y, c);
52  }
```

Listing 10. Multidimensional adaptation sample in Korz, extracted from (Ungar 2014)

colour and coordinate spaces, whereas the Korz example can perform arbitrary functional mappings.

B. Listings From Original Samples

In this section we reproduce the original sources of referenced examples from other papers.

B.1 Dimensional Adaptation in Korz

In section 9 we rework an example from (Ungar 2014) exhibiting freely dimensioned context awareness. In listing 10 we reproduce the original Korz language version as seen in (Ungar 2014):

B.2 Class Hierarchy Inheritance in Newspeak

In section 8 we rework an example from (Bracha 2013) exhibiting what is there named class hierarchy inheritance.

```
1   class ShapeLibrary usingPlatform: platform = (
2       | We use = to define immutable slots.
3       private List = platform collections List.
4       private Error = platform exceptions Error.
5       private Point = platform graphics Point.
6   |
7   )
8   (
9       public class Shape = (...)(...)
10      public class Circle = Shape (...)(...)
11      public class Rectangle = Shape (...)(...)
12  )
13
14  class ExtendShapes withShapes: shapes = (
15      | ShapeLibrary = shapes. |
16  )(
17      public class ColorShapeLibrary usingPlatform: platform =
18          ShapeLibrary usingPlatform: platform (
19      )(
20          public class Shape = super Shape ( | color | )(...)
21      )
22  )
```

Listing 11. Class hierarchy inheritance sample in Newspeak, extracted from (Bracha 2013)

In listing 11 we reproduce the original Newspeak language version as seen in (Bracha 2013).

References

John Backus *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Communications of the ACM Volume 21 Issue 8, pages 613-641, 1978.

Antranig Basman *If What We Made Were Real – Against Imperialism and Cartesianism in Computer Science*, Proceedings of the 28th Annual PPIG Workshop, 2017.

Antranig Basman, Luke Church, Clemens Klokmose, Colin Clark *Software and How it Lives On – Embedding Live Programs in the World Around Them*, Proceedings of the 27th Annual PPIG Workshop, 2016.

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.

Frederick P. Brooks,Jr., The Mythical Man-month (Anniversary Ed.), Addison-Wesley, 1995.

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

James O. Coplien *Curiously Recurring Template Patterns* C++ Report: 24–27, 1995.

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.

Martin Fowler *Inversion of Control Containers and the Dependency Injection Pattern*, 2004 url-https://martinfowler.com/articles/injection.html

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.

Liskov, B. *Keynote address - data abstraction and hierarchy*. ACM SIGPLAN Notices. 23 (5): 1734, 1988

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.