The Open Authorial Principle

Supporting Networks of Authors in Creating Externalisable 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, context-oriented and dependency injection systems, and concluding with the most recent generation of highly dynamic systems such as Korz and Newspeak. We follow 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. This raises conceptual and practical parallels with technologies and idioms supporting the web, such as REST, realised document structures supported by the DOM, and the negotiated space of CSS selectors.

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 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 with object-orientation at the base level 1, and ending with reuse requirements at level 4 which appear to be economically beyond the capabilities of current systems.

[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.

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.

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* $\oplus$ 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 2. In this presentation of reuse levels, we will sometimes use the term "component" to refer to the design artefacts/elements referenced by the authors' expressions.

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 component 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 separate examples in sections 4.1 and 4.2). 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 a familiar Meyerian system, ECMAScript 6 classes. 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

[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.

```
// Author A, artefact alpha
class LocalFlowManager extends FlowManager {
    keyIn() {
        ....
    }
}

// Author B, artefact alpha'
class UntrustedFlowManager extends LocalFlowManager {
    keyIn() {
        ....
    }
}

// Author E consumes alpha or alpha'
var FlowManagerTester = function(underTest) {
    ...
}
// Author F executes test
FlowManagerTester(new LocalFlowManager());
```

Listing 1. A basic exercise of reuse in ECMAScript 6

base class or *interface* here named a simple "flow manager" (FlowManager) and deliberately left out of scope. A vast literature on the nature of this substitutability 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'. This situation is illustrated in row 1 of table 2. Any base contract on substitutability which the language permits (FlowManager) is named for this discussion $\aleph$ (aleph), which may also act as a base implementation artefact.

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 following section 3.2. In a more open system, we would hope to shift the burden of adaptation either from F to E or indeed any further author in the system.

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 to an "untrusted data source" gpii.dataSource.untrusted[4]. Listing 3 shows the rendering of this situation as ECMAScript classes. In this listing, 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 2, 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 explicitly 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.

[4] In this case "untrusted" via metonymy signifies "a data source suitable to act in a case where the user does not trust the security of the local device"

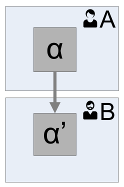
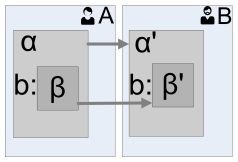
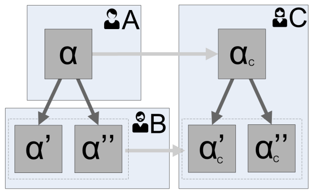
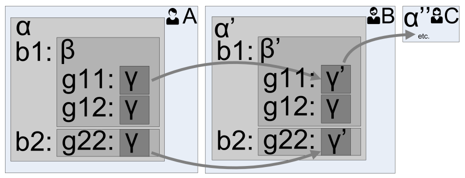
Level	Example Scenario	Systems Treating	Section	Diagram of Example Scenario
Level 1	A single artefact α created by A is extended to α' by B	Object Orientation	Section 2	
Level 2	A’s α with a nested artefact β has β extended to β' by B, creating an overall α'	Parameterised Types, DI, AOP	Section 3	
Level 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	Beta, Newspeak and AOP (wide hierarchy), Korz (all)	Section 4.1, Section 4.2	
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	. . .	Section 4.3	 A α b1: β g11: γ g12: γ b2: g22: γ B α’ b1: β’ g11: γ’ g12: γ b2: g22: γ’ α’’ C etc.

Table 2. Table of levels of increasingly sophisticated reuse with illustrations

```
// Author A, artefact alpha
class LocalFlowManager extends FlowManager {
    constructor: {
        this.preferencesDataSource = new LocalDataSource();
    }
}

// Author B, artefact beta'
class UntrustedDataSource;

// Author B now must also write artefact alpha'
class UntrustedFlowManager extends LocalFlowManager {
    constructor: {
        this.preferencesDataSource = new UntrustedDataSource();
    }
}

// Author E consumes alpha or alpha'
var FlowManagerTester = function(underTest) {
    ...
}
// Author F executes test - must also select alpha or alpha'
FlowManagerTester(new UntrustedFlowManager());
```

Listing 3. Economic failure of level 2 reuse in ECMAScript 6

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-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 every 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 module’s 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 T> class alpha: public T {}
```

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 `T` 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 `T` 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. (Basman 2017) follows through this argument in more detail.

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 *pointcut* 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 pointcuts 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 pointcuts 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[5], 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

- 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,

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

```
class ShapeLibrary usingPlatform: platform = (
    | "We use = to define immutable slots".
    private List = platform collections List.
    private Error = platform exceptions Error.
    private Point = platform graphics Point.
|
)
(
    public class Shape = (...)(...)
    public class Circle = Shape (...)(...)
    public class Rectangle = Shape (...)(...)
)

class ExtendShapes withShapes: shapes = (
    | ShapeLibrary = shapes. |
)(
    public class ColorShapeLibrary usingPlatform: platform =
        ShapeLibrary usingPlatform: platform (
    )(
        public class Shape = super Shape ( | color | )(...)
    )
)
```

Listing 4. Class hierarchy inheritance sample in Newspeak, extracted from (Bracha 2013)

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 2, and exampled in section 4.3. New facilities are needed to respond to these situations.

4.1 Level 3 Reuse Variant - Class Hierarchy Inheritance

Note that level 3 reuse scenarios may be demanding on account of two orthogonal kinds of forces. The first kind of force stresses designs where B has extended A's design into a deep hierarchy, and C wishes to advise all of it. Bracha (2010) names this variety of level 3 reuse as the "class hierarchy inheritance" problem. In the formulation there, author A has created a base class `Shape`, author B a `ShapeLibrary` deriving `Rectangle`, `Circle`, etc. and author C has created a hierarchy of colorable things, and wishes to make all the contents of B's library of shapes available to E as coloured shapes without having to do work proportional to the size of the library. This situation is illustrated in row 3 of table 2, and in listing 4 we reproduce the original Newspeak language version as seen in (Bracha 2013), showing that this reuse scenario is compactly and idiomatically resolved.

4.2 Level 3 Reuse Variant - Independent Context Dimensions

Continuing from section 4.1, level 3 reuse scenarios may be demanding on account of a second kind of force, which stresses designs where there is a large number of authors, B, 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". Here we present an example from (Ungar 2014) which 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).

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 (B)
- An "Australian" user on whose behalf the image will be rendered upside down (D)
- A user from Antarctica on whose behalf the image should be rendered at double size as well as upside down (E)

As an factorisation artefact, we label the expressions of an intermediate author C as mediating the expressions of D and E by creating a common context representing the "Southern Hemisphere", `southernHemi`.

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.

In listing 5 we reproduce the original Korz language version as seen in (Ungar 2014). In this listing we see that the expressions from different authors are somewhat functionally separate and do not intrude on the base artefacts (newCoord, screen). For example, the expressions of author B, "colour blind" appear on lines 29-33, those of author D, "Australian" on line 25, and those of author E, "Antarctica" on lines 26 and 49-52. Note that the expressions of authors

```
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 5. Multidimensional adaptation sample in Korz, extracted from (Ungar 2014)

C and B have become correlated together into the joint definition on line 40-47 expressing how to draw pixels on behalf of users who are both colour-blind and in the Southern Hemisphere, which morally would have to be the work of a higher-level integrator F who has to have become aware of the work of both C and B and synthesizes them in this way. In theory this intermediate definition might be unnecessary in a more powerful authorial system that could observe that the effects of these two adaptations commute and could factorise the dispatch `drawPixel` into two separate operations.

4.3 A Level 4 Reuse Scenario

We conclude our series of reuse examples from sections 2.1 and 3 with a more complex example of level 4 reuse. In this author network, author A (originally B in section 2.1) implemented an "untrusted `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 wished to write an integration test which verified the compatibility of A and B's work by aggregated them back into a single local design. C needs to direct overriding configuration at a nested part of B's 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 6 we see a rendering of this relationship in Newspeak, a modern language offering a powerful reuse mechanism first seen in Beta, named "virtual classes".

As we can see, the definitions for author C have become deeply nested, in order to address the structure of the deeply nested artefacts inherited from A and B. This doesn't represent economic reuse since the depth of this nesting will increase with the size of the pool of authors and their activities. In addition, we could interpret this situation as representing a form of violation of the "Law of Demeter" in which authors should not depend on direct structural knowledge of the layout of objects designed by others.

One might argue that authors A and B should have factored their definitions more appropriately, arranging to surface the piece of configuration which would eventually need to make its way to C at each stage of reuse. However, as in section 3.3, we are emphasising the fact that, under the OAP, effective reuse should not necessarily depend on such design anticipation and that we should be able to make some kind of economic use of previous authors' expressions purely through additionality.

```
"TODO: Currently some kind of cod-Newspeak"

class LocalDataSource = () ()
class UntrustedDataSource = () ()
class FlowManager = (
    | public preferencesDataSource = LocalDataSource new: |
) ()
class LocalFlowManager = FlowManager () ()
class UntrustedFlowManager = LocalFlowManager (
   | public preferencesDataSource = UntrustedDataSource new: |
) ()
class CloudBasedFlowManager = FlowManager () ()
class CloudBasedConfig = (
   | public flowManager = CloudBasedFlowManager new: |
) ()
class LocalConfig = (
   | public flowManager = LocalFlowManager new: |
) ()

"C's expression:"
class MultiConfig = (
   |
   public localConfig = LocalConfig new:.
   public cloudBasedConfig = CloudBasedConfig new:.
   |
) (
   public class CloudBasedConfig = super CloudBasedConfig(
       | public flowManager = CloudBasedFlowManager new: |
   )
   (
       public class CloudBasedFlowManager = super CloudBasedFlowManager (
           | public preferencesDataSource =
               LocalDataSource new: url: 'http://localhost:8088/preferences/%userToken'.
           |
       )
       ()
   )
)
```

Listing 6. FlowManager expression showing level 4 reuse requirement

Our preferred means of resolving such problems appeals to what could be called "query-based extension". We would prefer that author C was able to issue a query structured as a selector, which would specify in logical terms the address of the nested definition they wished to adapt, without having to precisely anticipate each layer of containment along the way to it. In this way, the "algebra of differences" would enable C to compactly encode differences between program designs in a moderately stable way. Providing capability, however, has cascading implications for many aspects of system and language design that we will begin to elaborate in section 6.

Firstly, however, we return to our principle in the light of these examples and consider some more direct aspects of its interpretation and implications for the economics of development.

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 name of an component 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 4.1, 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.

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. Addressibility and Externalisability

Following our presentation of an algebra of program differences in section 1.2, we now discuss two closely related properties that we argue must emerge in the design of a successful openly authorable (OA) system. These can be derived from the contrast we draw in section 5.4 between our algebra and similar such algebras that have been assembled in the past. Rather than merely a formal algebra whose elements may only be composed from a "God's eye view" amongst the elements themselves, we are concerned with an algebra whose affordances are available in practice to the network of authors, and in particular the user E we incorporated into this collection in section 1. To start with, this implies that we take on board all the goals of the Live Programming movement, epitomised by (Ungar 2013)'s slogan "The thing on the screen is supposed to be the actual thing". However, the nature of the algebra immediately implies a quite different structure for our implementations than has led the Live Programming community to implementations such as Smalltalk, Self and Korz.

In order to operate this algebra in an economically effective manner, we must allow externalisable references from outside an executing system to bind to instances of components in a way which is relatively stable across the variations in design induced by the contribution of author expressions. Our algebra cannot be operated with crude, unstable source-level coordinates such as program line numbers directing where differences are taken from and where they are projected to. We aim to recast the work of programming in terms of a *natural coordinate system* in which units of design have meaningful, stable names which identify their location in a fine-grained tree of cells within the design.

This aim leads to the following two properties of components in an OA design:

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

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

6.1 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, 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. OA 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:

6.1.1 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. These could be structured very similarly to the CSS system. A further precedent for such selectors binding to tree-structured elements is the regular-expression-like SMARTS language for encoding predicates on molecular graphs (Daylight 2008). A bridging analogy considering chemical reactions as a computational model appeared in (Berry 1992). It is these selectors which form the basis for the "query-based adaptation" model that we propose for achieving level 4 reuse in section 4.3.

6.1.2 Lifecycle of interactions aligned with component 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. A successful OA system needs to behave similarly, and prevent any further dispatch 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. These requirements would be weakened in a system which adopted a fully asynchronous message-passing idiom as is seen, for example, in Erlang.

In a general-purpose OA system, rather than one which simply represents a display document structure as the DOM, 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 adaptations into the tree at large. All of these relationships must be cleanly torn down when the component is destroyed. A realisation of this kind of multi-lateral relationship in such a context appears in the *Entanglements* of (Basman 2018).

6.2 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 constitutive 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. Drawing up the protocol for such messages also creates a great burden for tracking and interpretation by the communicating systems, a problem referred to by (Lanier 2003) as operating "simulations of vast tangles of telegraph wires".

As we identified in section 6.1, 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 6.1 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. A successful OA system, similarly, places state (and not message passing) at its architectural core and facilitates architectures that work with it. This implies that any messages transmitted between parts of a system preferentially should have the interpretation of either transmitting state wholesale, or else have a direct interpretation as transmitting a differential between two previously synchronised bodies of state.

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.

7. Dynamic Adaptation and Dispatch

Following our arguments section 6 on the necessarily cellular structure of successful openly authorable systems, we continue to make yet further indirect deductions from our principle about the strategies such systems should most likely use in place of familiar primitives from object-oriented and live systems.

A central feature of every object-oriented system, and those from many other traditions, is its algorithm for*dispatch*. This is the means by which the runtime selects amongst multiple available choices for the implementation of a method based on the environment around the call site. Both of the highly adaptible level 3 systems that we surveyed in section 4 solved their reuse questions through use of a highly dynamic dispatch — that is, the runtime performs a complex calculation at the point where an operation is invoked, in order to determine which implementation should be selected. We argue that strongly satisfying the OAP implies that we take a different approach tand satisfy the requirements of externalisability by operating a largely static dispatch.

Our externalisability obliges us to work with highly heterogeneous architectures, cooperating across machine, process, language and platform boundaries. Rather than distributing agency via the common "proxy" model that we describe in section 6.2, we recommend instead what we term the "avatar" model, described in (Clark 2017), whereby a portion of one system becomes a fully effective representative for a portion of another, for some bounded period of time. This is conformant with the REST model discussed in that section, where entire resources are transferred, rather than answers to limited questions. As a result of this heterogeneity, we are guided towards models of dispatch that are likely to be intelligible and computable in the widest variety of environments; an environment with highly dynamic dispatch cannot be emulated in one without it. As an example, we may find ourselves with highly undynamic peers implemented in languages like C or GLSL shaders.

How could we expect to implement highly dynamic, context-aware overall applications whilst locally maintaining our commitment to strong externalisability? Drawing again from the fund of techniques inspired by the web technologies discussed in earlier sections, we propose a system based on an equivalence between *wholesale adaptation* and *differential adaptation*. In just the same way as we propose in section 6.2 to interpret messages passed between systems as either wholesale or differential transfers of state, we propose that adaptations of a system be available in both differential and wholesale varieties, where the latter is expensive but can be honoured in environments with poor dynamism, and the former is cheap but requires dynamic, complex runtimes. Under wholesale adaptation, an appreciable part, or the entirety of an implementation is torn down, the dynamic state it accrued during its runtime temporarily stored elsewhere, and it is then rebuilt in the adapted form with the dynamic state reapplied. This model of adaptation is described in (Basman 2016), there named "Queen of Sheba adaptation".

There are a few requirements on the implementation in order to make this form of adaptation practical. Firstly, there must be a filtration of its state which allows the dynamic portion of it to be characterised and separated during the period of destruction. This requirement is likely to be met naturally as a result of meeting the requirements for strong externalisability that the OAP already implies. Secondly, there needs to be a capability to prevent observation of the system during the time the wholesale adaptation is in progress. Traditional idioms for such capabilities involve transactions or atomic operations of similar kinds which are familiar from mature persistence technologies.

Closing the circle on our discussion of web technologies, these transactions are also familiar from naturally evolved implementations of the DOM in modern web browsers. Web developers will be familiar with the phenomenon whereby, if a portion of the DOM is destroyed and re-rendered sufficiently promptly, the rendered page will appear to adapt itself continuously without the intervening destruction becoming apparent. Web browsers of recent years in fact opportunistically allocate "transactions" to batch together quickly successive updates to the DOM so that the user experience can be of a continuous adaptation. In the kinds of OA systems we seek to build, such transactions will have to be explicitly surfaced as primitives of the overall system and its protocols.

This scheme of allowing successive static systems to ape a dynamic one can be compared to the dynamic >= static idiom of (Edwards 2013), or some of the techniques seen within the Generative Programming tradition (Czarnecki 2000).

8. 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 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 reuse capabilities, some of which we articulate in this paper, and others which remain to be explored. A key implication of the principle is the elimination of horizons in the graph of authors, by allowing program differences to be freely expressed and combined as programs. This algebra gives rise to the fundamentally different architectural strategies required for successful openly authorable systems. These strategies also promote more natural externalisation of designs, supporting more straightforward interactions with external systems implemented in different processes, languages and idioms.

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, Philip Tchernavskij, Simon Bates and Michel Beaudouin-Lafon *An Anatomy of Interaction: Co-occurrences and Entanglements*, ⟨Programming⟩, Proceedings of Salon des Refusés Workshop, 2018.

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.

Gérard Berry and Gérard Boudol, *The chemical abstract machine*, Theoretical computer science, 96:1, 217–248, 1992.

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 1-10, 2005.

Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publ. Co., New York, NY, USA.

Daylight Chemical Information Systems, Inc. *SMARTS - A Language for Describing Molecular Patterns*, `http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html`, 2008.

Jonathan Edwards, *dynamic => static: type as subtext*, `https://vimeo.com/74314050`, 2013

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

J. Lanier *Why Gordian software has convinced me to believe in the reality of cats and apples* edge.org, November, 1. `https://www.edge.org/conversation/jaron_lanier-why-gordian-software-has-convinced-me\\-to-believe-in-the-reality-of-cats`, 2003.

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`, 2003.

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): 17–34, 1988

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

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

David Ungar and Randall B. Smith *The thing on the screen is supposed to be the actual thing* `http://davidungar.net/Live2013/Live_2013.html`, 2013.

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.